# Objektorientierte Programmierung

## Kapitel 8: Anweisungen (Statements)

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2013/14

http://www.informatik.uni-halle.de/~brass/oop13/

# Inhalt


1. Allgemeines
2. Blöcke
3. Bedingte Anweisungen
4. Schleifen
5. Sprünge

# Statements: Allgemeines (1)

- Anweisungen (engl. Statements) sind Teile eines Programms, die den Zustand verändern.

  Z.B. Variablen einen neuen Wert zuweisen, oder eine Ein-/Ausgabe vornehmen.

- Die sukzessive Änderung des Berechnungszustands ist in imperativen Sprachen das zentrale Programmierkonzept.

  Java ist objektorientiert, aber das bezieht sich nur auf die Struktur von Programmcode und Daten. Innerhalb jeder Methode ist es imperativ. Funktionale/logische Sprachen haben dagegen keine änderbaren Variablen.

- In C/C++/Java können auch Werteausdrücke (Expressions) den Zustand verändern, hier ist die Trennung zu Statements nicht so klar wie z.B. in der Sprache Pascal.

  Bei einem Wertausdruck ist die Berechnung des Wertes die Hauptsache, Zustandsänderungen kommen nur als Seiteneffekt vor. Statements liefern keinen Wert, die Zustandsänderung ist ihr einziger Zweck.

# Statements: Allgemeines (2)

- Die Bedeutung (Semantik) eines Statements ist in der Hauptsache eine Abbildung von Berechnungszuständen (Variablenwerten etc.) auf Berechnungszustände, z.B.

Alter Zustand

| Variable | Wert |
|---|---|
| i | 7 |
| n | 3 |

→ Statement: `n = n + i;` →

Neuer Zustand

| Variable | Wert |
|---|---|
| i | 7 |
| n | 10 |

- Wenn man mehrere Statements $S_1 \ldots S_n$ nacheinander schreibt, werden diese normalerweise in der gegebenen Reihenfolge abgearbeitet, so dass der Ergebniszustand von $S_i$ der Ausgangszustand von $S_{i+1}$ ist.

# Statements: Allgemeines (3)

- Es ist allerdings möglich, dass die Ausführung eines Statements "abrupt" endet, und zwar aufgrund
  - einer Exception (Ausnahmefall/Fehler, z.B. Division durch 0),
  - einer `return`, `break` oder `continue`-Anweisung (s.u.).
- Dann wird die normale Auswertungsreihenfolge verlassen, und zu einem übergeordneten Punkt in der Programmausführung gesprungen.
- In Sprachen wie Pascal gibt es diese Möglichkeit nicht: Dort hat jedes Statement genau einen Eingang und genau einen Ausgang.

  In Java gibt es zwar auch nur einen Eingang (man kann nicht in ein Statement hinein springen), aber mehrere Ausgänge: Je nach Grund für das abrupte Ende wird die Ausführung an unterschiedlichen Stellen fortgesetzt.

# Expression Statement (1)

- In Java sind bestimmte Ausdrücke als Anweisungen zulässig:
  - Zuweisung: ⟨Variable⟩ = ⟨Ausdruck⟩ (auch mit `+=`, etc.)

    Anweisungs-Bestandteil (z.B. rechts vom "=") können alle Ausdrücke sein.
  - Pre/Post-Inkrement/Dekrement-Ausdruck, z.B. `Var++`
  - Methodenaufruf, z.B. `obj.m(...)` oder `Class.m(...)`

    Die Methode muss nicht den Ergebnistyp `void` haben. Der Ergebniswert würde dann allerdings nicht verwendet, was etwas fragwürdig ist.
  - Objekterzeugung, z.B. `new Class(...)`

    Eine Objekterzeugung kann auch ohne Zuweisung an eine Variable Sinn machen, wenn der Konstruktor das neue Objekt in eine Datenstruktur einträgt, über die es zugreifbar bleibt.
- Der Ausdruck wird jeweils zur Anweisung, indem er mit einem Semikolon "`;`" abgeschlossen wird.

# Expression Statement (2)

- Ein Ausdruck wie

```
n + 1; // Syntaxfehler!
```

  kann in Java nicht als Statement verwendet werden: Er würde einen Wert berechnen, der gleich wieder vergessen wird.

  In C/C++ kann man jeden Ausdruck als Statement verwenden, dort wäre es legal (aber sinnlos). Wenn man die Warnungen beim C++-Compiler anschaltet, würde er darauf hinweisen, dass dieses Statement "no effect" hat.

- Folgendes wäre dagegen legal, aber ebenso sinnlos:

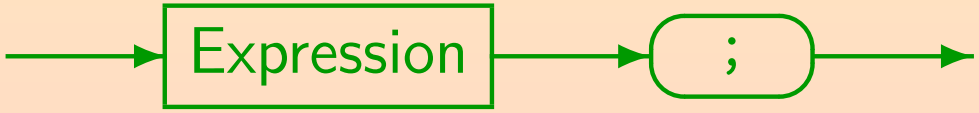
```
Math.sin(Math.PI/6);
```

  Der Java-Compiler prüft Methoden/Funktions-Aufrufe nicht, ob sie eine Zustandsänderung bewirken. Ein C++-Compiler bietet normalerweise die Möglichkeit, eine Warnung auszugeben, wenn der von einer Funktion gelieferte Wert ignoriert wird. Will man den Wert bewusst ignorieren, kann man einen Cast nach `void` schreiben. In Java geht das nicht (weil `void` kein Typ ist). Java-Compiler liefern aber auch keine solchen Warnungen.

# Expression Statement (3)

- Wenn man von einem Ausdruck zu einer Anweisung übergeht,
  - spielt der berechnete Wert keine Rolle mehr (er wird "vergessen"),
  - wichtig ist allein die Zustandsänderung ("der Seiteneffekt").
- Syntaxdiagramm:

Expression Statement:




Die Grammatik im Hauptteil der Java-Spezifikation stellt sicher, dass nicht beliebige Expressions möglich sind, sondern nur die oben genannten.
Die Grammatik im Anhang, die wirklich zur Implementierung des Compilers verwendet wurde, tut das nicht (so wie dieses Syntaxdiagramm).
Der Compiler muss dann in seiner semantischen Analyse testen, dass die Expression von einer der obigen Formen ist.

# Leere Anweisung

- So wie Ausdrücke selbst wieder Ausdrücke als Bestandteile enthalten können, so können auch Anweisungen aus kleineren Anweisungen aufgebaut sein (z.B. bei `if`, `while`).
- Manchmal braucht man die syntaktisch verlangte Teilanweisung aber nicht (wenn z.B. der Ausdruck in der Bedingung der Schleife schon alles macht).
- Deswegen ist es gelegentlich praktisch, dass auch ein einzelnes Semikolon eine Anweisung ist.

Empty Statement:

- Diese Anweisung tut nichts (keine Zustandsänderung).

# Inhalt

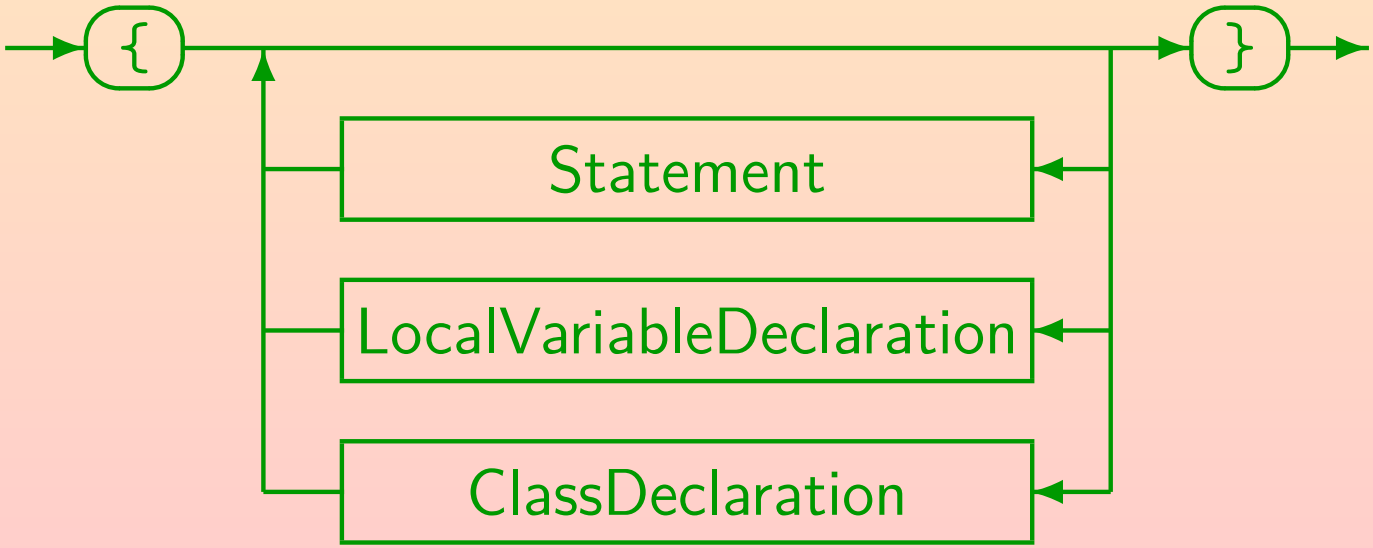

1. Allgemeines
2. Blöcke
3. Bedingte Anweisungen
4. Schleifen
5. Sprünge

# Block/Sequenz (1)

- Eine Folge von Anweisungen und Deklarationen, eingeschlossen in geschweifte Klammern "{" und "}", ist wieder eine Anweisung ("block").

  In Java sind (im Gegensatz zu C++) Deklarationen nicht überall als Statements erlaubt. Java schließt so sinnlose Deklarationen aus, z.B. als einzelne abhängige Anweisung eines `if`. Die Variable könnte nicht mehr verwendet werden.

- Block:



# Block/Sequenz (2)

- Durch die Zusammenfassung von Statements als Block kann man an Stellen, an denen syntaktisch nur eine Anweisung erlaubt ist (z.B. die von `if` abhängige Anweisung) eine ganze Folge von Anweisungen unterbringen.
- Die in einem Block zusammengefassten Anweisungen werden sequentiell nacheinander ausgeführt.

  D.h. erst wird die erste Anweisung vollständig ausgeführt, dann die zweite, u.s.w.
- Wenn eine der Anweisungen abrupt endet (durch eine Exceptionen oder einen Sprung mit `break` etc.), endet die Ausführung des ganzen Blocks entsprechend.

  Die folgenden Anweisungen werden dann nicht mehr ausgeführt.

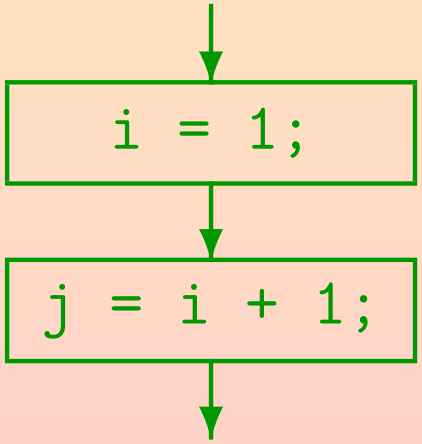
# Block/Sequenz (3)

- Z.B. kann man

```
{ i = 1; j = i + 1; }
```

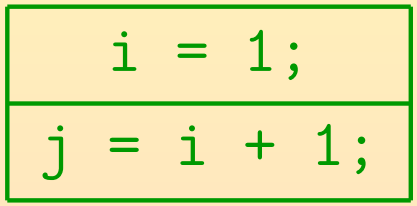
graphisch als Flussdiagramm so veranschaulichen:

Statt Flussdiagramm sagt man auch "Programmablaufplan".



# Block/Sequenz (4)

- Als Alternative zu Flussdiagrammen gibt es auch Struktogramme (Nassi-Shneiderman-Diagramme):




Flussdiagramme sind in DIN 66 001 genormt, Struktogramme in DIN 66 261. Flussdiagramme sind in Verruf geraten, weil sie unstrukturierten Programmen mit beliebigen Sprüngen entsprechen. Sie scheinen mir persönlich aber übersichtlicher zu sein (und zeigen eher, wie die CPU das Programm abarbeitet).

Wenn ich Algorithmen (Berechnungsverfahren) erläutern will, verwende ich aber Pseudocode: Das ist Text, der übliche Kontrollstrukturen wie die von Java verwendet, aber Teilschritte nur natürlichsprachlich (in Deutsch) beschreibt, so dass er für die direkte Ausführung noch nicht geeignet ist, aber dem menschlichen Leser auf einer etwas höheren Abstraktionsebene erklärt, was zu tun ist.

# Block/Sequenz (5)

*

- In manchen anderen Sprachen (z.B. Pascal) schreibt man
  - "`begin`" statt "`{`",
  - "`end`" statt "`}`".

    C hatte schon immer einen Hang zur Kürze.
- In Pascal wird das Semikolon zur Trennung der Anweisungen verwendet, in C/C++/Java schließt es Anweisungen ab.

  D.h. in Pascal würde nach der letzten Anweisung kein "`;`" stehen.
- In C++ und Java können Statements und Deklarationen beliebig gemischt werden, früher (z.B. in C) war dagegen die Reihenfolge "erst Deklarationen, dann Statements" üblich.

  Die neue Regelung ist praktisch, weil man am Anfang des Blocks nicht immer einen sinnvollen Wert zur Initialisierung der Variablen kennt.

# Variablen-Deklarationen (1)

- Eine Deklaration teilt dem Compiler (und dem Leser des Programms) mit, wofür ein Bezeichner steht, der im Programm verwendet wird.

  Der Compiler muss ja Speicherplatz für die Variable reservieren.

  Die Anzahl Bytes hängt vom Datentyp ab.

- Beispiel für Variablendeklaration (ohne Initialisierung):

  ```
  int i;
  ```

  Dies deklariert eine Variable mit Namen "`i`", in die ein Wert des Typs "`int`" gespeichert werden kann.

  Man kann auch kurz sagen: "Es wird eine `int`-Variable `i` deklariert".

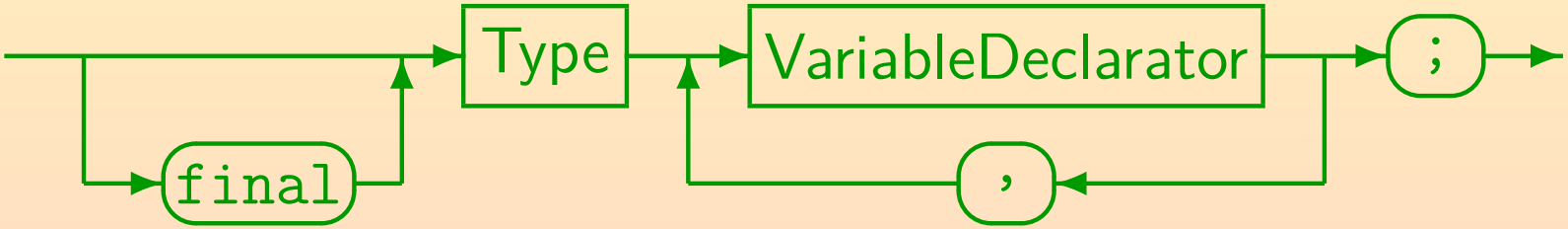
  Beispiel für Variablendeklaration mit Initialisierung: `int i = 0;`

- In Blöcken deklarierte Variablen heißen "lokale Variablen", um sie von in Klassen deklarierten Variablen ("Attributen") zu unterscheiden.

# Variablen-Deklarationen (2)

- Eine Variablendeklaration kann mehrere Variablen des gleichen Typs deklarieren, die einzelnen Variablen (ggf. mit Initialisierung) werden durch Komma getrennt:

  LocalVariableDeclaration:



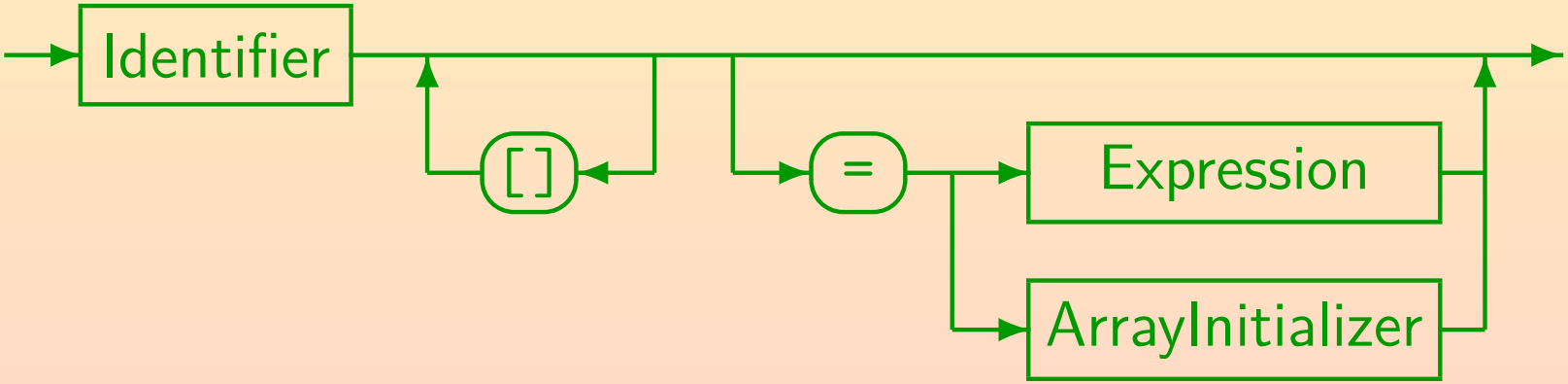

- Das optionale Schlüsselwort "`final`" bedeutet, dass man der Variablen nur einmal einen Wert zuweisen kann.

  Wenn man die Variable z.B. gleich initialisiert, kann es später keine weitere Zuweisung an sie geben, d.h. dies ist der endgültige Wert. Falls man sie mit einen konstanten Ausdruck initialisiert, ist sie selbst eine Konstante. Falls man sie nicht gleich initialisiert, muss der Compiler später bei der Zuweisung sicher sein, dass es vorher keine Zuweisung gegeben hat.

# Variablen-Deklarationen (3)

- Nach dem Typ folgt in der Variablendeklaration ein Bezeichner für die neue Variable und optional ein Wert zur Initialisierung (berechnet durch beliebige Expression):

  VariableDeclarator:




- Für Arrays gibt es eine Sonderbehandlung (siehe Kapitel 17):
  - Statt "`int[] a;`" kann man auch "`int a[];`" schreiben.
  - Initialisierung z.B. "`int[] a = {1, 2, 3};`".

# Variablen-Deklarationen (4)

- Deklarationen müssen textuell vor der Verwendung der Variablen stehen.

  Der Compiler muss erst die Deklaration sehen, bevor er Code für Zugriffe auf die Variable erzeugen kann. Der Typ der Variablen bestimmt ja die zu erzeugenden Maschinenbefehle, und eventuell nötige Typumwandlungen.

- Der Gültigkeitsbereich einer lokalen Variablen-Deklaration erstreckt sich auf den Rest des Blockes, in dem die Variablen-Deklaration steht.

  Inklusive dem Ausdruck zur eigenen Initialisierung und den Initialisierungen aller danach noch deklarierten Variablen. Inklusive auch geschachtelter Blöcke.

- D.h. dort ist die Variable bekannt und kann verwendet werden.

  Am Ende des Blockes, in dem die Variable deklariert wurde, wird die Variable wieder vergessen/gelöscht.

# Variablen-Deklarationen (5)

- Innerhalb einer Methode kann man nicht zwei lokale Variablen mit gleichem Namen deklarieren, so dass eine Deklaration innerhalb des Gültigkeitsbereiches der anderen steht.

  Wenn man den Namen verwendet, muss ja klar sein, auf welche Deklaration er sich bezieht. (Für Experten: Innerhalb einer geschachtelten Klassendeklaration wäre es möglich, eine Variable gleichen Namens einzuführen.)

- Eine lokale Variable darf auch nicht so heißen wie ein Parameter der Methode.

- Eine lokale Variable darf dagegen so heißen wie ein Attribut.

  Die Verwendung von Attributen (Variablen in Objekten) wurde in Kapitel 6 diskutiert, die Deklaration eigener Attribute besprechen wir später im Rahmen der Klassendeklarationen. Wenn lokale Variable und Attribut gleich heißen, wird das Attribut "verschattet", d.h. ist nicht mehr durch einfache Angabe des Namens zugreifbar: Dieser bezieht sich dann auf die lokale Variable.

# Variablen-Deklarationen (6)

- Die Initialisierung einer lokalen Variable geschieht jedesmal neu, wenn die Deklaration "ausgeführt" wird.
- Beispiel:

```
(5)     int i = 1;
(6)     while(i <= 5) {
(7)         int n = 1;
(8)         if(n > 1)
(9)             System.out.println("Huch!");
(10)        n++;
(11)        i++;
(12)    }
```

- Die Ausgabeanweisung wird nie ausgeführt, weil `n` beim `if` immer den Wert 1 hat.

# Uninitialisierte Variablen (1)

- Eine Deklaration wie z.B.

  ```
  int i;
  ```

  reserviert zwar Speicherplatz für die Variable `i`, aber trägt in diesen Speicherplatz keinen Wert ein.

- Solange der Variablen noch kein Wert zugewiesen ist, nennt man sie "uninitialisiert".

  Im Hauptspeicher stehen natürlich irgendwelche Nullen und Einsen, die aber kaum vorhersehbar sind: Vorher stand an der Stelle vielleicht eine Variable von ganz anderem Typ, die jetzt nicht mehr benötigt wird.

- Der Java-Compiler akzeptiert nur Programme, bei denen beweisbar kein Lese-Zugriff auf eine uninitialisierte Variable vorkommt.

  D.h. bevor man den Wert einer Variablen abfragt, muss man erst einen Wert eintragen (durch Zuweisung an die Variable).

# Uninitialisierte Variablen (2)

- Z.B. ist nach folgendem Programmstück klar, dass `i` initialisiert ist:

```
(5)   int n = scanner.nextInt();
(6)   int i;
(7)   if(n > 0)
(8)       i = 1;
(9)   else
(10)      i = 0;
(11)  System.out.println(i); // ok
```

- Der Compiler erkennt, dass in beiden Zweigen der Fallunterscheidung `i` ein Wert zugewiesen wird. Also hat `i` danach einen definierten Wert.

  Bei einer Zuweisung im Rumpf einer Schleife ist dagegen nicht sicher, dass die Schleife auch nur einmal ausgeführt wird.

# Uninitialisierte Variablen (3)

- Leider können Compiler nicht immer feststellen, ob Zugriffe auf uninitialisierte Variable vorkommen.

  Dies hängt ja auch von den Eingabewerten ab. Man kann beweisen, dass diese Frage unentscheidbar ist, d.h. dass es unmöglich ist, ein Computerprogramm zu schreiben, das andere Computerprogramme auf die korrekte Initialisierung der Variablen für beliebige Eingaben prüft (und immer nach endlicher Zeit eine richtige Antwort ausgibt).

- Wenn der Java-Compiler sich nicht sicher ist, dass eine Variable initialisiert wurde, gibt er eine Fehlermeldung aus.

  Es gibt kein "im Zweifel für den Angeklagten". Wenn der Compiler also keine Fehlermeldung ausgibt, sind die Variablen sicher initialisiert. Zugriffe auf uninitialisierte Variablen können in Java nicht vorkommen.

# Uninitialisierte Variablen (4)

- Bei folgendem Programmstück gibt der Java-Compiler eine Fehlermeldung wegen uninitialisierten Variablen aus, obwohl es das Gleiche tut, wie das obige `if-else`:

```
(5)     int n = scanner.nextInt();
(6)     int i;
(7)     if(n > 0)
(8)         i = 1;
(9)     if(n <= 0)
(10)        i = 0;
(11)    System.out.println(i); // Fehler!
```

Die Fehlermeldung lautet:

```
Test.java:10: variable i might not have been initialized
        System.out.println(i);
                           ^
```

- Lösung: Programm umschreiben, notfalls eigentlich überflüssige Initialisierung direkt bei der Deklaration.

# Uninitialisierte Variablen (5)

- Nur der Zugriff auf eine nicht sicher initialisierte Variable ist ein Fehler. Das folgende Programmstück wird durch den Compiler akzeptiert:

```
(5)     int i;
(6)     if(n > 0) {
(7)         i = 1;
(8)         System.out.println(i);
(9)     }
(10)    System.out.println("Ende");
```

- Wenn auf `i` in Zeile (8) zugegriffen wird, ist sie sicher initialisiert. Es spielt keine Rolle, dass sie in Zeile (10) nicht mehr unbedingt initiaisiert ist.

  Wenn die Variable `i` nur im Block unterhalb des `if` benutzt wird, sollte man sie allerdings auch dort deklarieren.

# Uninitialisierte Variablen (6)

- Bei C/C++ ist der Programmierer selbst dafür verantwortlich, dass er nicht auf uninitialisierte Variablen zugreift.

  Falls es doch vorkommt, ist es ein schwierig zu findener Fehler: Eventuell bemerkt man es überhaupt nicht und bekommt falsche Ausgabewerte. Das Programm wird ja übersetzt und ohne Laufzeitfehler vollständig ausgeführt (Der Zugriff auf eine uninitialisierte Variable ist nicht wie eine Division durch 0: Man bekommt einfach den Wert, der dort zufällig im Hauptspeicher steht).

- Allerdings kann man Compiler-Warnungen anschalten, die ähnlich wie der Test in Java funktionieren (sollte man tun).

- In C und C++ werden Arrays nicht automatisch initialisiert.

  Die Prüfung der Initialisierung jeder einzelnen Element-Variable in einem Array ist für heutige Compiler zu kompliziert. Java stellt durch automatische Initialisierung sicher, dass es hier auch keine Zugriffe auf uninitialisierte Variablen gibt. Das kostet aber Laufzeit. Hier liegt der eigentliche Unterschied.

# Inhalt


1. Allgemeines
2. Blöcke
3. Bedingte Anweisungen
4. Schleifen
5. Sprünge

# If Statement (1)

- Mit der `if`-Anweisung ist es möglich, andere Anweisungen nur beim Vorliegen bestimmter Bedingungen auszuführen (englisch "if": "wenn", "falls").

  Die `if`-Anweisung gehört zusammen mit der `switch`-Anweisung zur Klasse der bedingten Anweisungen.

- Die `if`-Anweisung gibt es in zwei Varianten:
  Mit und ohne `else` ("sonst", "andernfalls").

- Beispiel:

```
(5)     if(x >= 0)
(6)         y = x;
(7)     else
(8)         y = -x;
```

# If Statement (2)

- Die `if`-Anweisung hat folgenden Aufbau:

```
if(Bedingung)
    Statement1
else
    Statement2
```

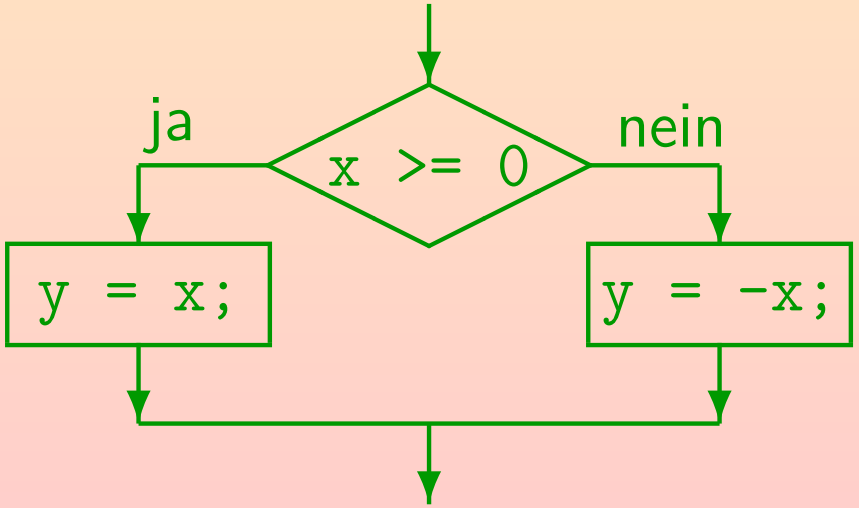
- Dabei ist "Bedingung" ein Wertausdruck (Expression) vom Typ `boolean` oder `Boolean` (später: "Wrapperklasse").
- Es wird zunächst die Bedingung ausgewertet.
- Ist sie `true` (wahr), wird die vom `if` abhängige Anweisung $\text{Statement}_1$ ausgeführt.
- Ist sie `false` (falsch), so wird die vom `else` abhängige Anweisung $\text{Statement}_2$ ausgeführt (sofern vorhanden).

# If Statement (3)

- `if` bewirkt eine Verzweigung im Programmablauf.
- Beispiel (nochmals):

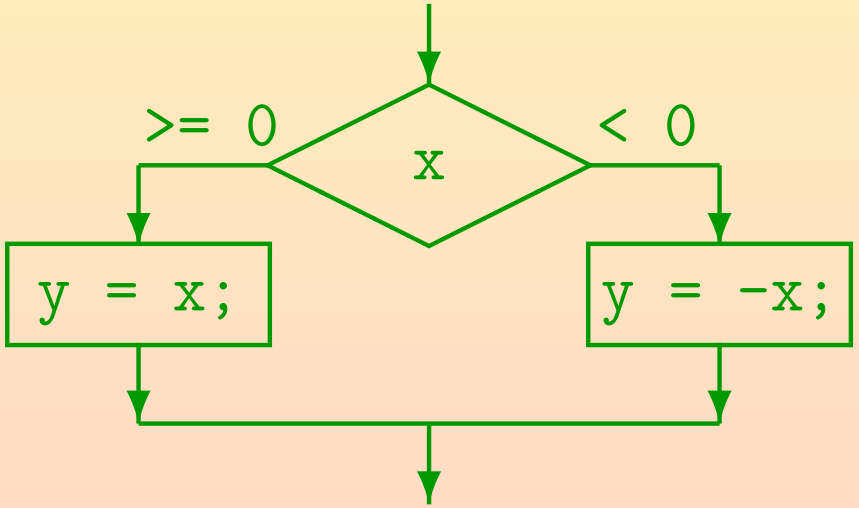
```
(5)     if(x >= 0)
(6)         y = x;
(7)     else
(8)         y = -x;
```

- Flussdiagramm:



# If Statement (4)

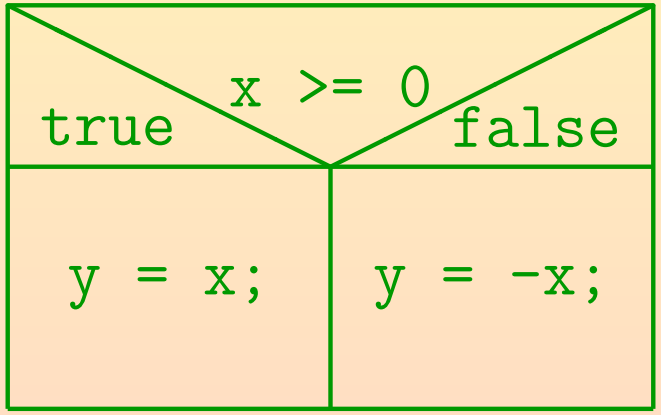
- Flussdiagramm (Alternative):




Die Bedingungskanten müssen die Raute nicht immer links und rechts verlassen. Auch auf einer Seite und unten ist möglich.

# If Statement (5)

Struktogramm:




Statt "`true`" und "`false`" kann man auch "`T`" und "`F`" schreiben.

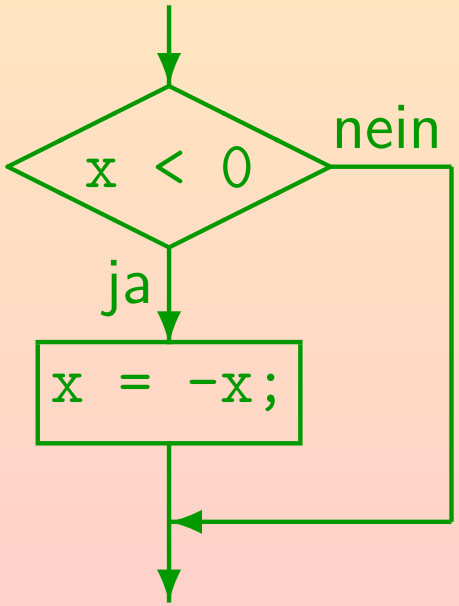
Manche Autoren schreiben die Bedingung auch "`if(x >= 0)`".

# If Statement (6)

- Beispiel (`if` ohne `else`):

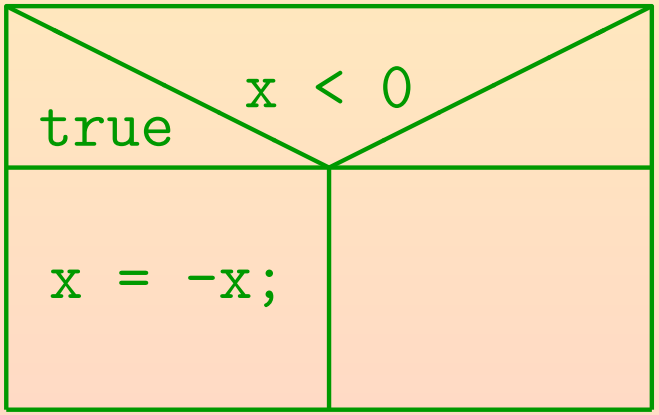
```
if(x < 0)
    x = -x;
```

- Flussdiagramm:



# If Statement (7)

- Struktogramm für den Fall "`if` ohne `else`":



# If Statement (8)

- Die vom `if` abhängige Anweisung wird manchmal auch die `then`-Klausel genannt, weil man in Pascal und ähnlichen Sprachen folgendes geschrieben hat:

```
if x < 0 then
        x = -x
```

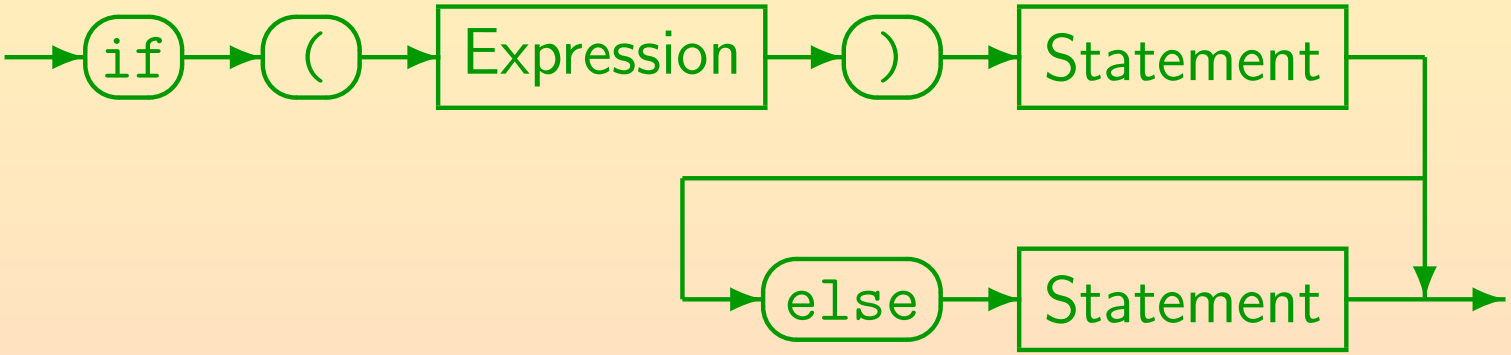
**Kein Java!**

- Hier braucht man keine Klammern um die Bedingung (`if` und `then` sind sozusagen die Klammern).
- In C/C++/Java muss die Bedingung dagegen immer in Klammern eingeschlossen sein.

# If Statement (9)

- Syntaxgraph:

  IfStatement:




- `else` gehört immer zum nächstmöglichen `if`:

  `if (A) if (B)` $S_1$ `else` $S_2$

  wird verstanden als

  `if (A) { if (B)` $S_1$ `else` $S_2$ `}`

  und nicht als

  `if (A) { if (B)` $S_1$ `} else` $S_2$

# If Statement (10)

- Weil die `if`-Anweisung selbst wieder eine Anweisung ist, lassen sich `else if`-Ketten bilden, indem man eine `if`-Anweisung für den `else`-Zweig einsetzt:

```
if(x > 0)
        sign = +1;
else if(x == 0)
        sign = 0;
else // x < 0
        sign = -1;
```

In jedem `else`-Zweig gilt, dass alle vorherigen `if`-Bedingungen falsch sind. Man kann ggf. einen Kommentar benutzen, um dies zu erklären.

# If Statement (11)

- Falls man in einem der Zweige mehrere Anweisungen sequentiell nacheinander ausführen will, muss man sie mit `{ ... }` zu einem Block zusammenfassen.

  Wie oben erläutert, nutzt eine korrekte Einrückung nichts.

- Selbstverständlich darf man immer `{ ... }` setzen (auch bei nur einer abhängigen Anweisung).

  Dem geübten Programmierer erscheint das aber eher umständlich.

- In der Sprache Algol 68 war das Problem mit schließenden Schlüsselworten für alle Kontrollstrukturen gelöst, z.B. `if ... then ... else ... fi` (**kein Java!**).

  Entsprechend: `while ... do ... od` (Schlüsselwort gespiegelt/rückwärts).

# If Statement (12)

*

- Die Programmiersprache C hatte keinen booleschen Datentyp.
- In der `if`-Bedingung stand ein beliebiger Ausdruck:
  Die Zahl `0` und die `null`-Referenz (der `NULL`-Pointer) zählten als falsch, alles andere als wahr.

  C++ hat einen booleschen Datentyp (`bool`), aber sehr großzügige Typumwandlungen, so dass korrekter C-Code auch in C++ korrekt ist.
- Der Vorteil gegenüber Java ist, dass die Bedingungen in C/C++ manchmal kürzer sind.

  In Java muss man den Vergleich mit `0` oder `null` explizit schreiben.
- Der Nachteil gegenüber Java ist, dass z.B. bei Verwechselung von == mit = der Fehler deutlich schwerer zu finden ist.

  Es entsteht ja legales C, nur verhält sich das Programm nicht wie erwartet.

  Bessere Compiler bieten allerdings Warnungen für dies Fall an.

# Switch Statement (1)

- Hinweis: Falls Sie mit der Programmierung Schwierigkeiten haben, konzentrieren Sie sich zunächst auf `if` und `while`.

  Man sollte mit `if` gefestigt sein, bevor man zum `switch` weiter geht.

- Es ist manchmal nötig, viele verschiedene Werte für eine Variable getrennt zu behandeln, z.B.

```
if(tag == 1)
    text = "Montag";
else if(tag == 2)
    text = "Dienstag";
...
else if(tag == 7)
    text = "Sonntag";
else
    text = "FEHLER!";
```

# Switch Statement (2)

- Solche Situationen kann man übersichtlicher mit der `switch`-Anweisung formulieren (Beispiel s.u.).
- Die `switch`-Anweisung ist nicht unbedingt nötig:
  Man kann damit nichts machen, was man nicht auch mit `if`/`else if`-Ketten machen könnte.

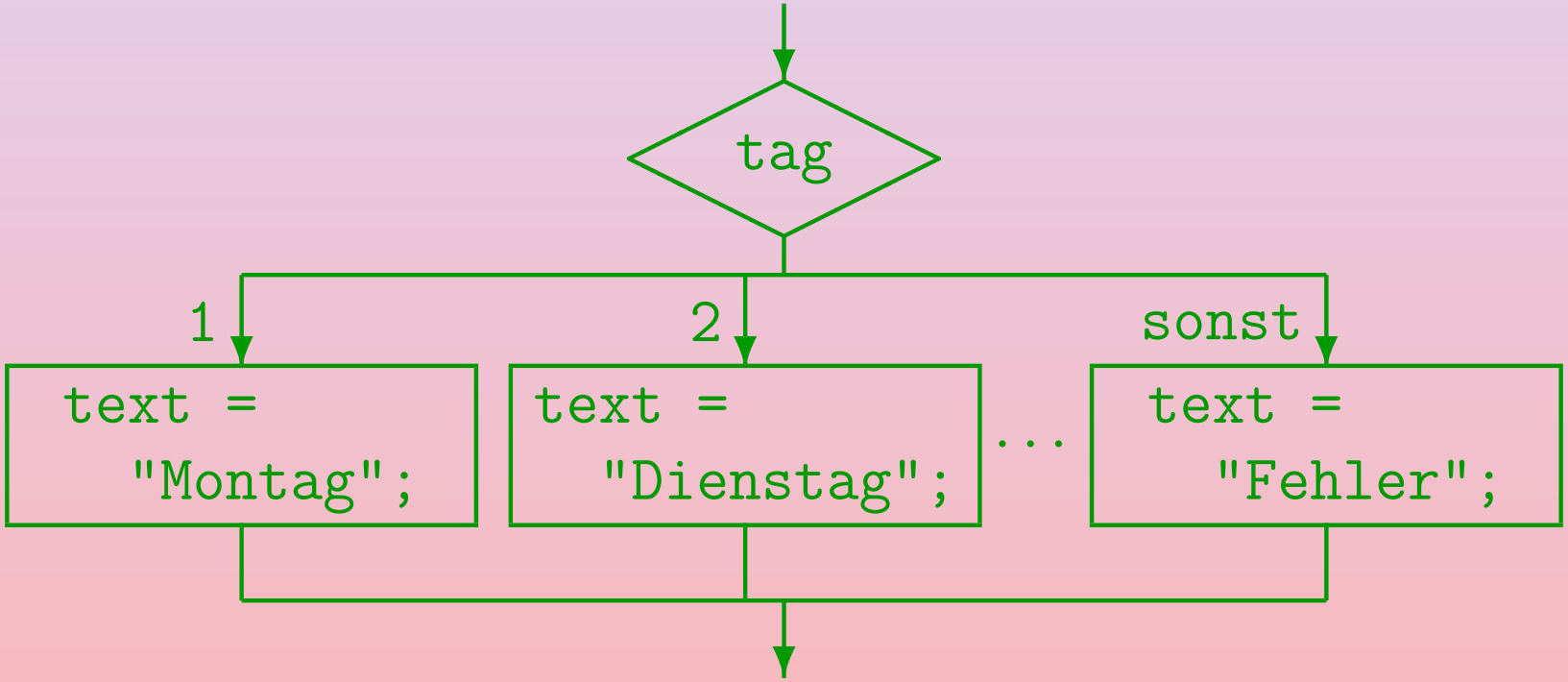
  Es würde also für den Anfang reichen, "`switch`" in seinem passiven Wortschatz zu haben (d.h. es lesen zu können). Man muss es nicht in eigenen Programmen verwenden (man outet sich dadurch aber als Programmier-Anfänger).
- Je nach Verteilung der zu betrachtenden Werte erzeugt der Compiler ggf. einen Sprung über eine Tabelle mit den Startadressen der verschiedenen Fälle.
- Das kann effizienter (schneller) sein als die entsprechende `if`/`else if`-Kette.

# Switch Statement (3)

```
switch(tag) {
    case 1:
        text = "Montag";
        break;
    case 2:
        text = "Dienstag";
        break;
    ...
    case 7:
        text = "Sonntag";
        break;
    default:
        text = "FEHLER!";
}
```

# Switch Statement (4)

Flussdiagramm:



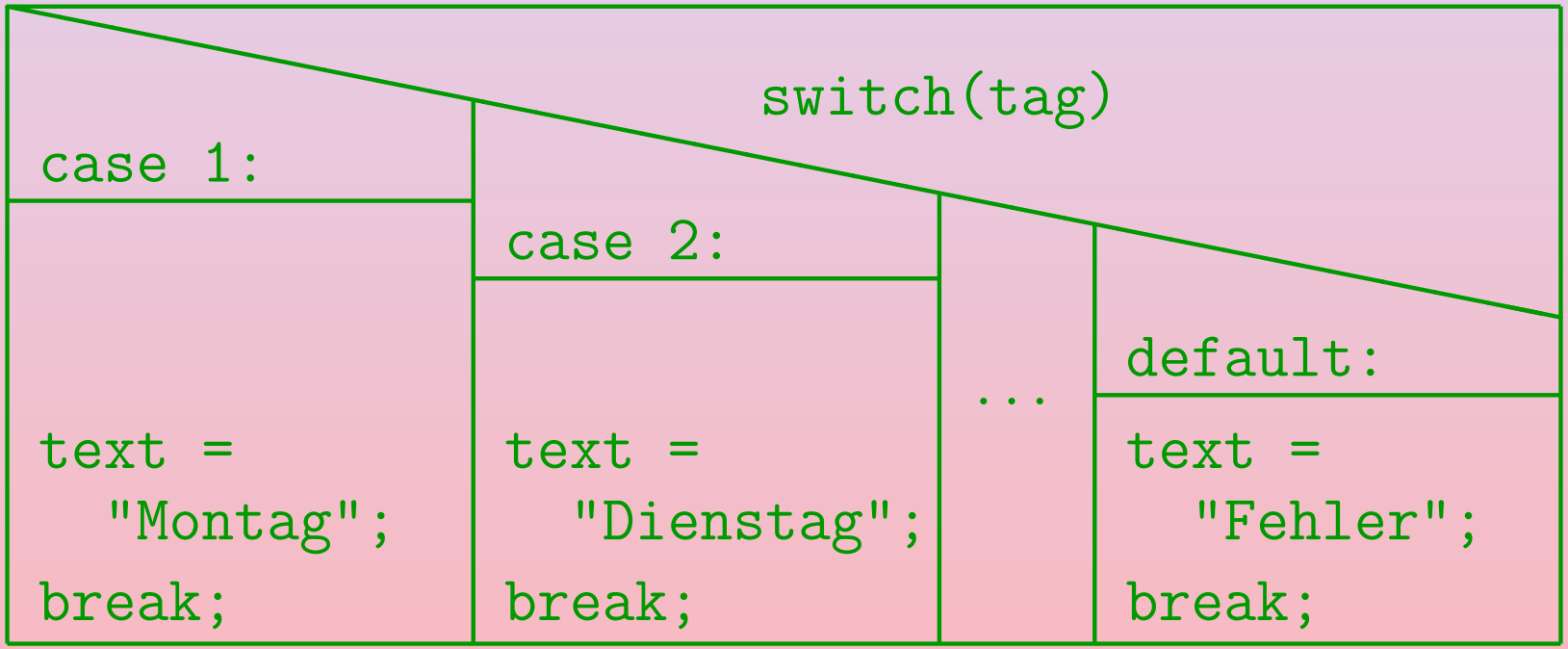
# Switch Statement (5)

Struktogramm:

| tag | | | |
|---|---|---|---|
| 1 | 2 | | default |
| text = "Montag"; | text = "Dienstag"; | ... | text = "Fehler"; |

# Switch Statement (6)

Struktogramm (Alternative):



# Switch Statement (7)

- Man kann auch mehrere verschiedene Werte in einem gemeinsamen Fall behandeln:

```
switch(c) {
    case 'a':
    ...
    case 'z':
        ... // Kleinbuchstabe
        break;
    case 'A':
    ...
    case 'Z':
        ... // Grossbuchstabe
        break;
}
```

# Switch Statement (8)

- Wie man am letzten Beispiel sieht, ist es nicht nötig, den `default`-Fall anzugeben.
- Sollte keiner der mit `case` angegebenen Fälle zutreffen, geschieht dann einfach gar nichts.

  D.h. es gibt ein implizites leeres `default`.
- Ein Programm sollte auch mögliche Fehlerfälle abfangen. Selbst wenn es eigentlich nicht passieren dürfte, dass keiner der `case`-Fälle zutrifft, sollte man einen `default`-Fall mit einer Fehlermeldung und ggf. einem Programmabbruch vorsehen.

# Switch Statement (9)

- Falls man das `break` am Ende eines Falls vergisst, wird die Ausführung mit den Anweisungen des folgenden Falls fortgesetzt.
- Das ist gelegentlich nützlich, aber in den meisten Fällen ein Fehler.

  Falls man es wirklich will, sollte man einen Kommentar verwenden, um darauf hinzuweisen, dass kein Fehler vorliegt. Z.B. `/* FALLTHROUGH */` oder `// Drops through.`
- Statt `break` kann man auch `return` verwenden, wenn man gleichzeitig die Prozedur beenden will.

# Switch Statement (10)

- Syntaxgraph:

  SwitchStatement:

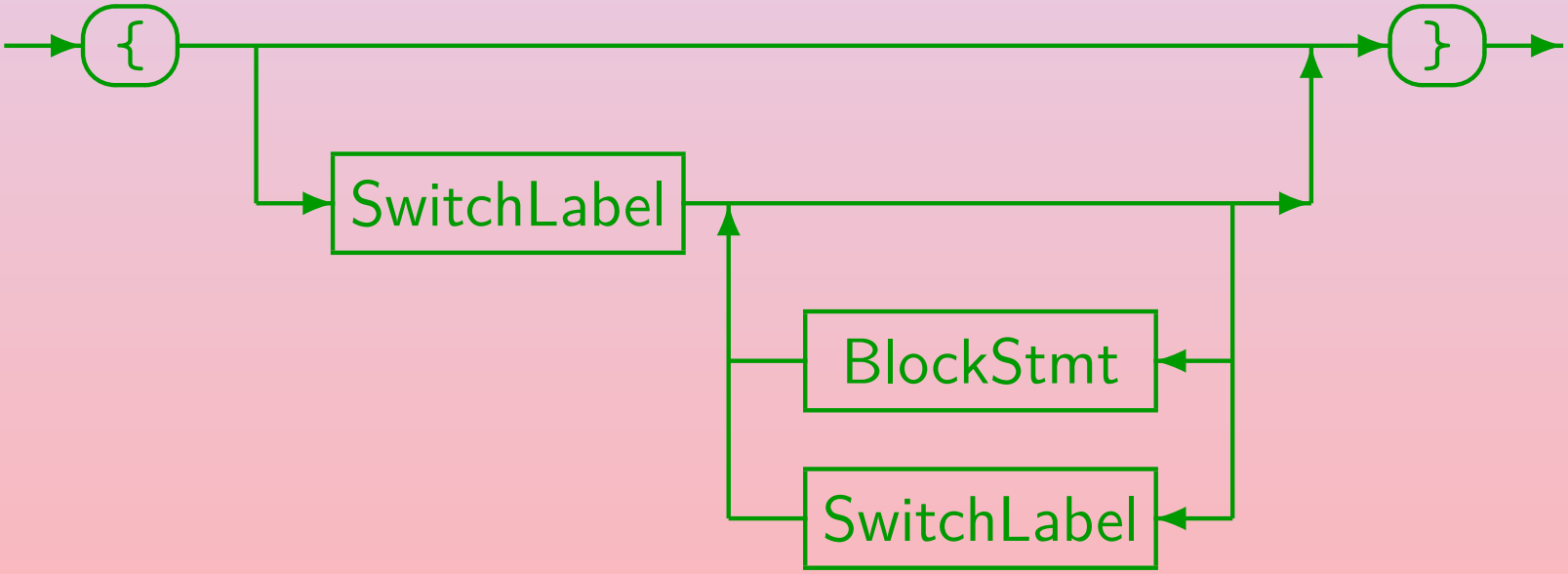
- Die Expression muss von einem der folgenden Typen sein: `char`, `byte`, `short`, `int`, `Character`, `Byte`, `Short`, `Integer`, ein Aufzählungstyp, oder, seit Java 7, `String`.

  Wenn man von Strings absieht, sind alles ganzzahlige Typen (bei Klassen wie `Integer` wird automatisch der in ihnen gespeicherte Wert ausgepackt). Für Experten: Strings werden intern über den Hashcode behandelt, also wird hier im wesentlichen auch mit ganzen Zahlen gearbeitet (natürlich wird nach der Verzweigung über den Hashcode geprüft, dass der String auch tatsächlich gleich ist, und ggf. Kollisionen behandelt).

# Switch Statement (11)

- Nach dem `switch` und dem Wert, über den gesprungen wird, kommt eine Art Block, der aber zusätzlich "SwitchLabels" (`case`, `default`) enthalten kann:

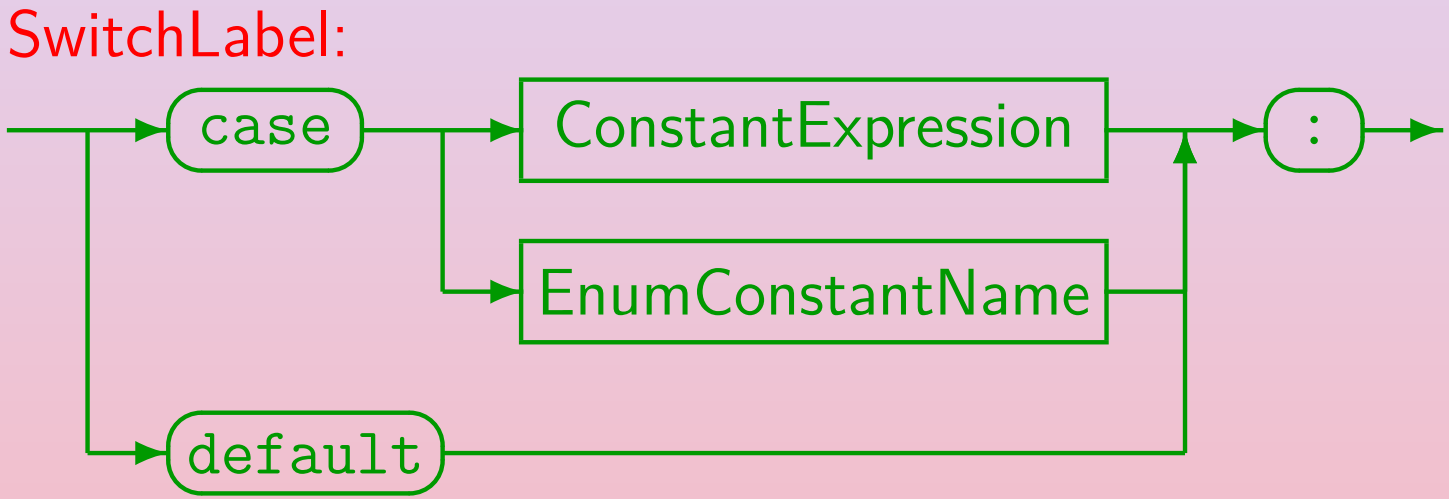
SwitchBlock:




"BlockStmt" ist alles, was in einem Block erlaubt ist (s.o.): Normale Statements sowie Deklarationen lokaler Variablen und Deklarationen lokaler Klassen.

# Switch Statement (12)

- Die einzelnen Fälle im `switch` werden mit `case` oder `default` markiert:




- Natürlich darf es in einem `switch` nicht zwei `case`-Fälle geben mit dem gleichen Wert, und auch nicht zwei `default`-Fälle.

  Für einen gegebenen Wert der `switch`-Expression muss eindeutig festgelegt sein, wo die Ausführung fortgesetzt wird.

# Switch Statement (13)

- Nach dem `case` muss eine "Constant Expression" stehen. Normalerweise ist das ein Datentyp-Literal (eine Konstante wie 123 oder `'a'`), aber man kann auch `+`, `-`, etc. anwenden.

  Die meisten Operatoren sind in konstanten Ausdrücken erlaubt, aber natürlich nicht Zuweisungen und `++`, `--`, auch nicht `instanceof`.

- Der Compiler muss in der Lage sein, den Wert einer "Constant Expression" zu berechnen.

  Das ist wichtig, damit er ggf. eine Sprungtabelle aufbauen kann.

- Normale Variablen sind in konstanten Ausdrücken natürlich verboten (ihr Wert steht erst zur Laufzeit fest).

- Symbolische Konstanten wie `Math.PI` sind dagegen erlaubt.

  Diese heissen in der Spezifikation "constant variables": `final`-Variablen mit Inititialisierung durch eine "ConstantExpression".

# Inhalt


1. Allgemeines
2. Blöcke
3. Bedingte Anweisungen
4. Schleifen
5. Sprünge

# While-Schleife (1)

- Die `while`-Anweisung führt eine abhängige Anweisung (den Schleifenrumpf) solange aus, wie eine Bedingung erfüllt ist (englisch "while": u.a. "solange wie").
- WhileStatement:

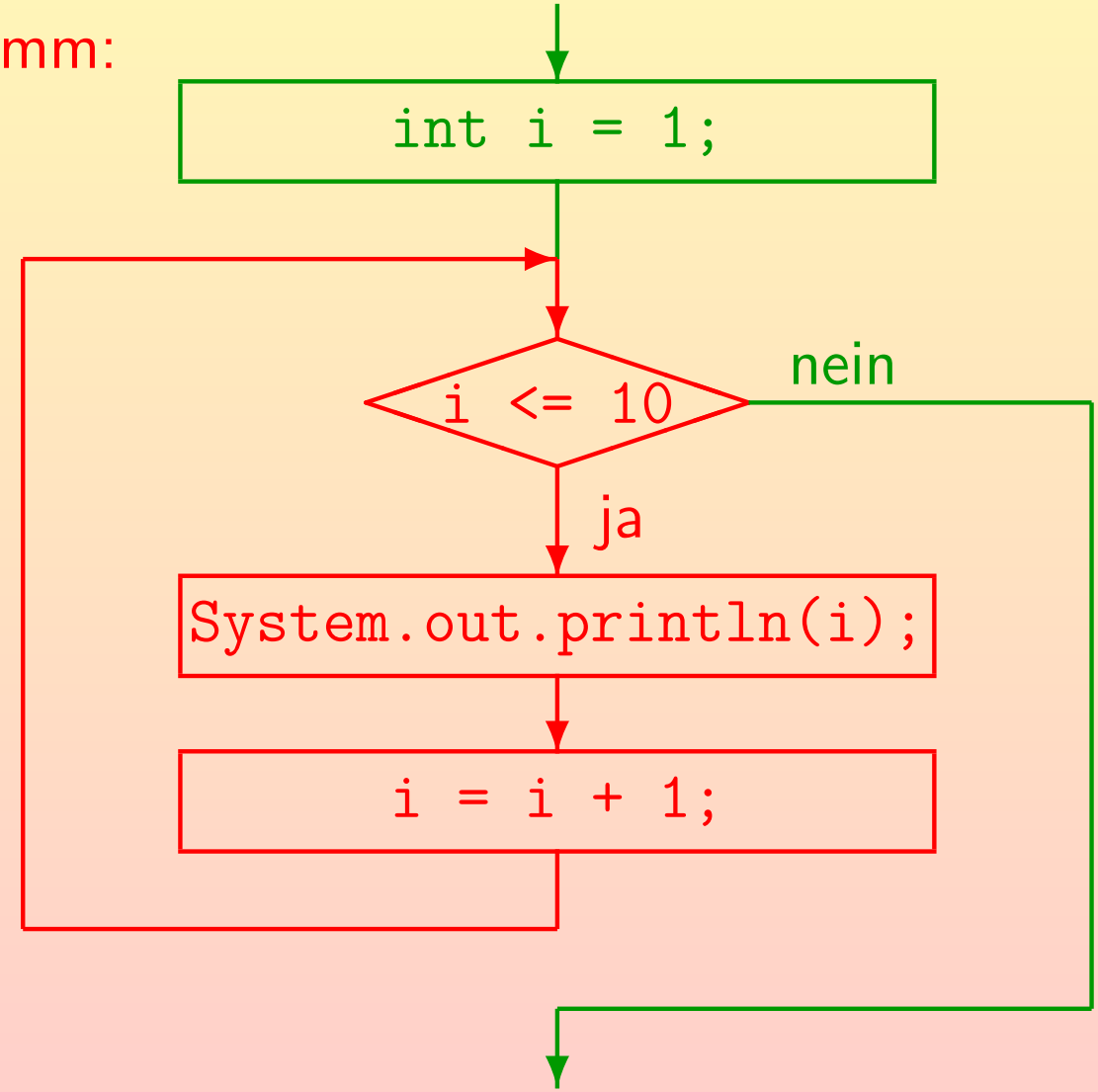
Die Expression ist die Schleifenbedingung, das Statement der Rumpf der Schleife.

- Beispiel:

```
int i = 1;
while(i <= 10) {
    System.out.println(i);
    i = i + 1;
}
```

# While-Schleife (2)

Flussdiagramm:



# While-Schleife (3)

- Struktogramm:

```
i = 1;
while(i <= 10)
    System.out.println(i);
    i = i + 1;
```

In der Literatur werden für den Schleifenkopf unterschiedliche Notationen verwendet. Manche schreiben z.B. "`Solange`" oder "`DO`" anstelle von "`while`" und lassen die Klammern weg. Manche geben auch nur die Schleifenbedingung an. Da das gleiche graphische Symbol auch für die `for`-Schleife verwendet wird, ist dies aber etwas problematisch.

# While-Schleife (4)

Aufgabe:

- Was ist der Fehler in diesem Programmstück?

```
System.out.print("Bitte n eingeben: ");
int n = scanner.nextInt();
int fak = 1;
while(n > 0); // Vorsicht!
{
    fak = fak * n;
    n = n - 1;
}
```

Tipp: Schauen Sie sich die Zeile mit dem `while` genau an!

- Warum meldet der Compiler keinen Fehler?

  D.h. warum ist es legaler Java-Code? Tipp: Schauen Sie auf Folie 9 und 11.

- Was passiert, wenn man es ausführt?

# While-Schleife (5)

- Man beachte, dass eine Schleife auch 0 Mal ausgeführt werden kann (wenn die Schleifenbedingung gleich zu Anfang falsch ist).

  Falls man eine Variable im Rumpf der Schleife initialisiert, ist nach Ende der Schleife nicht sicher, dass sie initialisiert ist. Deswegen erlaubt der Java-Compiler in diesem Fall keinen lesenden Zugriff. Eigentlich sollte man so eine Variable erst im Rumpf der Schleife deklarieren, dann wäre sie außen gar nicht zugreifbar.

- Um die Korrektheit eines Programms zu prüfen, ist es wichtig, solche Extremfälle durchzuspielen.

  Wann sollte auch durchdenken, was passiert, wenn die Schleife genau ein Mal durchlaufen wird. Im allgemeinen Fall sind der erste und der letzte Durchlauf der Schleife wichtige Kandidaten für eine manuelle Simulation.

# While-Schleife: Terminierung (1)

- Die `while`-Bedingung muss irgendwann falsch werden, sonst erhält man eine Endlosschleife.
- Wenn ein Programm nicht (freiwillig) anhält, sagt man auch, das Programm terminiert nicht.

  Man kann das Programm natürlich immer explizit abbrechen. Wie das genau geht, hängt vom Betriebssystem ab. Meist reicht es, "Ctrl+C" (Steuerung-C) zu drücken. Unter Windows liefert Ctrl+Alt+Del eine Liste aller Prozesse, aus der man den abzubrechenden Prozess aussuchen kann. Ein Prozess ist ein in Ausführung befindliches Programm. Unter UNIX zeigt "`ps`" oder "`ps -ef`" eine Liste von Prozessen an, der Abbruch geht dann mit "`kill` ⟨`Prozess-ID`⟩" bzw. notfalls "`kill -9` ⟨`Prozess-ID`⟩". Bei einer Entwicklungsumgebung gelangt man möglicherweise in den Debugger, wenn man "Ctrl+C" drückt, dann kann man sogar sehen, an welcher Stelle sich das Programm gerade befunden hat. Ein Debugger ist ein Programm, das hilft, Fehler ("Bugs") in einem anderen Programm zu finden.

# While-Schleife: Terminierung (2)

- Es ist unentscheidbar, ob ein Programm irgendwann anhält.

  Dies ist das bekannte Halteproblem. Es wird in Vorlesungen über theoretische Informatik ("Automaten und Berechenbarkeit") ausführlich behandelt.

- D.h. es wird nie einen Compiler geben, der genau dann eine Fehlermeldung ausgibt, wenn eine Schleife nicht terminieren wird.

  Natürlich muss der Compiler selbst terminieren. Das Problem wäre selbst dann unentscheidbar, wenn man sich nur für eine feste Eingabe interessiert.

- Der Compiler könnte in offensichtlichen Fällen eine Warnung ausgeben, aber die wenigsten tun das.

# While-Schleife: Terminierung (3)

- Um zu beweisen, dass eine Schleife enden wird, kann man jedem Berechnungszustand eine ganze Zahl zuordnen, die
  - beim Beginn jedes Schleifendurchlaufs nicht negativ ist, und
  - bei jedem Schleifendurchlauf um mindestens 1 reduziert wird.
- Im Beispiel wäre das z.B. $10 - \texttt{i}$.

  Durch `i = i + 1;` wird der Wert von $10 - \texttt{i}$ bei jedem Schleifendurchlauf verringert. Durch die Schleifenbedingung `i <= 10` ist sichergestellt, dass der Wert nie negativ wird.
- Diese Zahl ist eine obere Schranke für die noch nötigen Schleifendurchläufe.

# While-Schleife: Terminierung (4)

Aufgabe:

- Was halten Sie von der Schleife in diesem Programmstück zur Berechnung der Fakultät ($n! = 1 * 2 * 3 * \cdots * n$)?

```
System.out.print("Bitte n eingeben: ");
int n = scanner.nextInt();
int fak = 1;
while(n != 0) {
    fak = fak * n;
    n = n - 1;
}
System.out.println("n! = " + fak);
```

Tipp: Was passiert bei der Eingabe negativer Zahlen?

# While-Schleife: Logik (1)

- Dieses Programm zum Zahlenraten ist unnötig kompliziert:

```
int zahl = (int) Math.floor(Math.random()*101);
System.out.print("Bitte Zahl 0 - 100 raten: ");
int geraten = scanner.nextInt();
while(geraten != zahl) {
    if(geraten == zahl)
        System.out.println("Erraten!");
    ...
}
System.out.println("Erraten!");
```

- Der Rumpf der Schleife wird nur betreten, wenn die Schleifenbedingung erfüllt ist. Daher kann das erste `if` niemals wahr sein! Man kann die Anweisung streichen.

# While-Schleife: Logik (2)

- Man sollte Variablen der Schleifenbedingung erst am Ende des Schleifenrumpfes ändern (z.B. Laufvariable weiterschalten), damit man vorher sich darauf verlassen kann, dass die Schleifenbedingung gilt (auch übliches Muster).
- Umgekehrt kann man nach Abschluss der `while`-Schleife sicher sein, dass die Schleifenbedingung falsch ist:

```
while(geraten != zahl) {
    ...
}
if(geraten == zahl)
    System.out.println("Erraten!");
```

- Die Bedingung im `if` nach der `while`-Schleife ist hier immer erfüllt: `if` streichen und direkt "`Erraten!`" ausgeben.

# Begriff: Iteration

- Die wiederholte Ausführung einer Anweisung oder eines Anweisungsblocks nennt man eine "Iteration" (Wiederholung).
- Schleifenanweisungen (`while`, `do`, `for`) sind iterative Anweisungen (engl. "iterative statements").
- Ein Verfahren, das auf einer Schleife basiert, heisst auch "iterativ".

  Besonders, wenn man den Unterschied zu einem "rekursiven" Verfahren betonen will (siehe Kapitel 11 über Methoden).

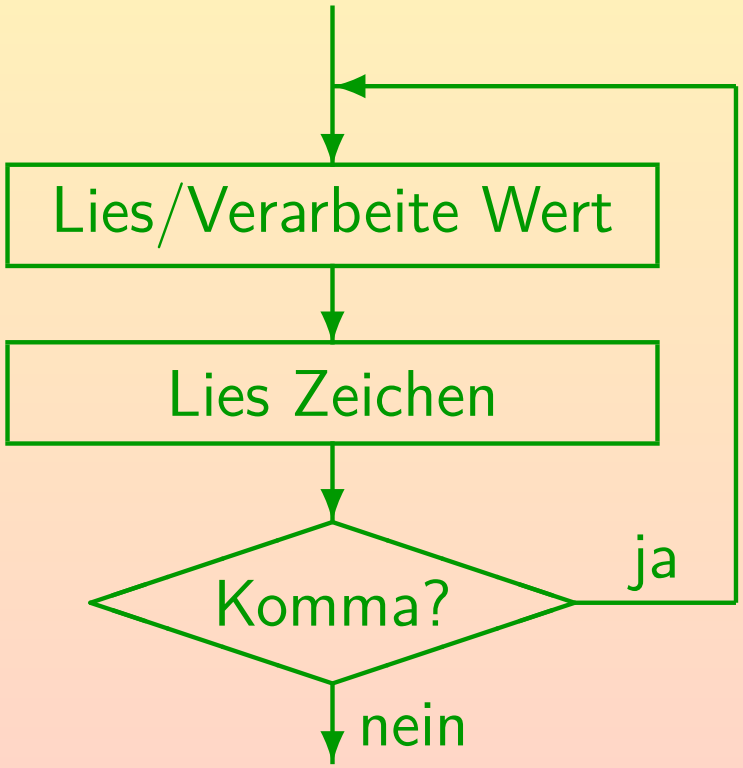
# Do-Schleife (1)

- Manchmal kommt es vor, dass erst am Ende des Schleifenrumpfes feststeht, ob die Schleife nochmal durchlaufen werden muss.

  Dann muss die Schleife natürlich auf jeden Fall mindestens einmal durchlaufen werden.

- Beispiel: Es soll eine durch Kommata getrennte Liste von Werten eingelesen werden (endet mit “.”).
  - Im Schleifenrumpf liest man jeweils einen Wert (und verarbeitet ihn).
  - Anschließend schaut man sich das nächste Zeichen an (Komma oder Punkt).

# Do-Schleife (2)



# Do-Schleife (3)

- Wenn man dieses Verfahren mit einer `while`-Schleife codieren will, muss man das Einlesen und Verarbeiten eines Wertes doppelt aufschreiben:
  - Vor der Schleife (für den ersten Wert), und
  - in der Schleife (für alle folgenden Werte).
- Verdoppelung von Programmcode ("Copy&Paste-Programmierung") ist immer schlecht:
  - Der Leser muss ein längeres Programm verstehen.
  - Wenn man etwas ändert, muss man immer beide Stellen ändern (vergißt man eine, wird es inkonsistent).

# Do-Schleife (4)

- Daher gibt es in C/C++/Java die do-Schleife:

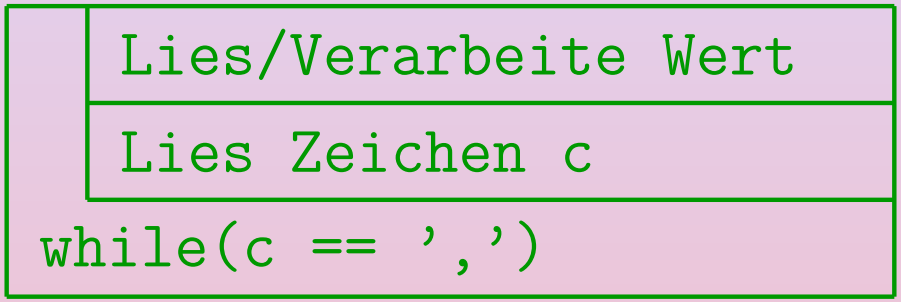
```
do {
        Lies/Verarbeite Wert; // Pseudocode
        Lies Zeichen c;       // Kein Java
} while(c == ',');
```

- Während die while-Schleife "kopfgesteuert" ist, ist dies eine "fußgesteuerte" Schleife.
- Der Rumpf der do-Schleife wird immer mindestens einmal durchlaufen.
- Diese Schleife wird nur sehr selten gebraucht!

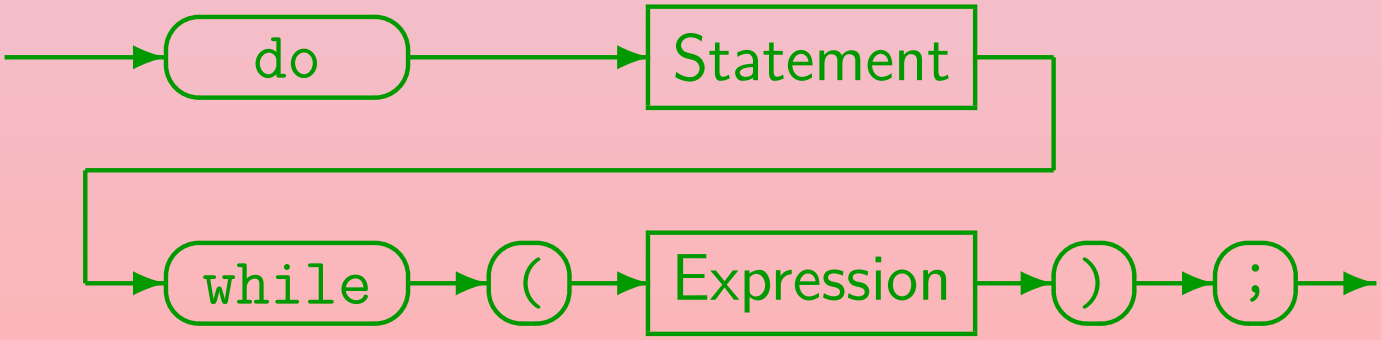
  Man kann damit nichts machen, was nicht auch mit der while-Schleife möglich wäre. Ich habe sie in 30 Jahren vielleicht 2–3 Mal verwendet.

# Do-Schleife (5)

Struktogramm:




Syntax-Diagramm (DoStatement):



# Do-Schleife (6)

- Viele Leute finden die Syntax des Do-Statements nicht besonders hübsch/übersichtlich.

  Man möchte die wichtige Schleifenbedingung lieber gleich oben sehen. Es besteht eine gewisse Verwechselungsgefahr mit einer `while`-Schleife mit leerem Rumpf, die in C/C++ durchaus vorkommt (deswegen sollte man hier nach "`}`" keinen Zeilenumbruch machen). In Pascal gibt es deswegen `repeat-until`, wobei hier aber verwirrend ist, dass eine wahre Schleifenbedingung zum Abbruch führt.

- Bjarne Stroustrup (Erfinder von C++) schreibt, dass nach seiner Erfahrung `do`-Schleifen häufiger zu Fehlern führen.

  Er sagt, dass auch für den ersten Durchlauf der Schleife (bevor die Bedingung geprüft wird), häufig doch etwas ähnliches wie die Schleifenbedingung gelten muss (damit der Rumpf korrekt funktioniert). Diese etwas abgeschwächte Bedingung wäre aber oft nicht garantiert.

# Do-Schleife (7)

- Vielleicht sollte man dem Rat von Herrn Stroustrup folgen und auf das do-Statement ganz verzichten.

  Oder es jedenfalls nur verwenden, wenn es eindeutig einen Vorteil bringt.

- Die Verdopplung von größeren Stücken Programmcode ist aber sicher schlimmer.
- Falls es sich aber nur um einen Ausdruck/eine Anweisung handelt, ist das unproblematisch.
- Dies kann man durch Einsatz von Methoden (s.u.) immer erreichen.

# For-Schleife (1)

- Die folgende Schleifenstruktur ist typisch:

```
i = 1;                  // Initialisierung
while(i <= 10) {        // Bedingung (Begrenzung)
        System.out.println(i);
        i = i + 1;      // Schritt (Weiterschalten)
}
```

- Man kann die Schleifensteuerung mit diesen drei Komponenten im Schleifenkopf zusammenfassen:

```
for(i = 1; i <= 10; i = i + 1)
        System.out.println(i);
```

- Die Programmstücke verhalten sich völlig gleich.

# For-Schleife (2)

- Die Variable `i`, die nacheinander eine leicht zu verstehende Folge von Werten annimmt, und damit den Rest der Schleife steuert, heißt auch die Laufvariable der Schleife.
- Wenn es eine Laufvariable gibt, ist die `for`-Schleife (nach einer gewissen Gewöhnung an die Syntax) übersichtlicher als die entsprechende `while`-Schleife.
- Man kann die `for`-Schleife aber als Abkürzung für die entsprechende `while`-Schleife definieren, sie gibt keine grundsätzlich neuen Möglichkeiten.

  Wenn Sie sich anfangs von den vielen Möglichkeiten überfordert fühlen, können Sie alle Schleifen auch nur mit `while` schreiben. Sie müssen die `for`-Schleife allerdings ggf. lesen können. Außerdem prägen sich Muster eventuell besser ein, wenn sie kürzer sind, hier hätte `for` einen Vorteil.

# For-Schleife (3)

- In Pascal sieht die `for`-Schleife so aus:

```
for i := 1 to 10 do
    writeln(i);
```

**Kein Java!**

- Die Syntax ist zunächst übersichtlicher.
- Außerdem terminiert diese Art der Schleife immer.

  In Pascal wird die `for`-Schleife immer so oft ausgeführt, wie die beiden Grenzen beim Start der Ausführung vorgeben. Man kann auf diese Art keine Endlosschleife bekommen.

- Die `for`-Schleife in C/C++/Java erlaubt dagegen auch ganz andere Arten von Laufvariablen: Nicht nur über Zahlen, sondern z.B. auch über Objekten in einer verketteten Liste.

  Die `for`-Schleife in Java ist also viel allgemeiner als die in Pascal, aber das hat seinen Preis. Verkette Listen werden später besprochen.

# For-Schleife (4)

- Es wäre ganz schlechter Stil, wenn die Laufvariable im Innern des Schleifenrumpfes geändert würde.

  Der einzige Vorteil der `for`-Schleife in C/C++/Java gegenüber der entsprechenden `while`-Schleife ist es, dass man die komplette Schleifenkontrolle gleich zu Anfang sehen kann. Es ist also klar, welche Werte die Laufvariable nacheinander annehmen wird. Denkt man jedenfalls.

  Eine Zuweisung an die Laufvariable im Schleifenrumpf würde diesen Vorteil ins Gegenteil verkehren: Der Leser rechnet damit nicht, sondern nimmt an, dass im Kopf der Schleife alles über die Laufvariable ausgesagt ist, was er wissen muss. Allenfalls könnte vielleicht ein `break`-Statement (s.u.) die Schleife vorzeitig beenden, aber auch das ist etwas problematisch (bei langen Schliefen eventuell in Kommentar ankündigen).

  In Pascal sind Zuweisungen an die Laufvariable im Schleifenrumpf verboten, in C/C++/Java wären sie legal.

# For-Schleife (5)

- Man kann die Laufvariable auch gleich in der `for`-Schleife deklarieren:

```
for(int i = 1; i <= 10; i = i + 1)
        System.out.println(i);
```

- Die Variable `i` ist jetzt nur innerhalb der Schleife bekannt (deklariert).

  Der Java-Compiler meldet einen Fehler, wenn man versucht, nach Ende der Schleife auf `i` zuzugreifen. Bei C++ war dies offiziell auch so, aber Microsoft Visual C++ erlaubte, auf die Variable noch nach der Schleife zuzugreifen. Solche Programme konnte man dann nicht mehr mit anderen Compilern übersetzen (z.B. mit dem GNU Compiler). Ähnliches passierte mit Java: 1997/98 gab es ein Gerichtsverfahren zwischen Sun und Microsoft, weil Microsoft eine inkompatible Version von Java verbreitete.

# For-Schleife (6)

- Die drei Teile der `for`-Schleife können (unabhängig von einander) entfallen:
  - Wenn die Laufvariable z.B. vorher schon initialisiert ist:

```
int n = scanner.nextInt();
if(n < 0)
        ...; // Fehlerbehandlung
for(; n > 0; n--)
        ...
```

  - Wenn das Weiterschalten der Laufvariable schon anders geschieht (so allerdings suboptimaler Stil):

```
for(int i = 0; i++ < 100; )
        ...
```

# For-Schleife (7)

- Optionalität der Komponenten der `for`-Schleife:
  - Läßt man die Bedingung weg, gilt sie als "true".
  - Es muss dann also eine andere Art geben, wie die Schleife beendet wird (z.B. eine `break`- oder `return`-Anweisung irgendwo im Schleifenrumpf).

    Dann hat man natürlich nicht mehr den Vorteil der `for`-Schleife, dass man die komplette Schleifenkontrolle sofort sieht. Immerhin ist aber offensichtlich, dass es ein `break` etc. geben muss.
  - Eine typische "for-ever"-Schleife ist

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

# For-Schleife (8)

- Es ist auch möglich, zwei Laufvariablen zu verwalten, z.B.

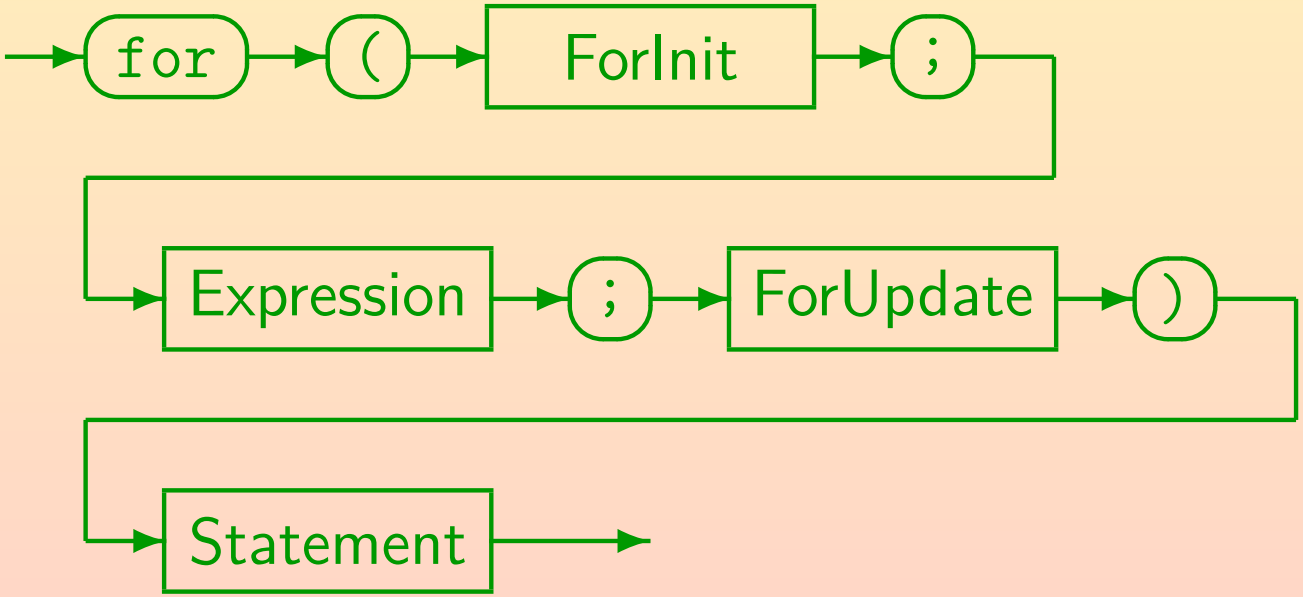
```
for(int i = 1, j = 100; i <= 100 && j > 0;
                                  i++, j--) {
      ...
}
```

Hier würde `i` von 1 bis 100 laufen, und `j` gleichzeitig von 100 bis 1. Dieses Beispiel hat keinen besonderen Zweck. Ein sinnvolleres Beispiel wäre das Durchlaufen einer verketteten Liste (später in der Vorlesung) mit gleichzeitigem Hochzählen einer Positionsnummer `i`. Es könnte aber sein, dass bei mehreren Laufvariablen eine `while`-Schleife übersichtlicher ist, oder man eventuell besser im `for` nur die eine Variable steuert.

In C++ kann man Ausdrücke mit dem Sequenzoperator "`,`" verknüpfen, das würde hier im Schritt-Teil verwendet. Java hat keinen allgemeinen Sequenzoperator, aber im ersten und dritten Teil der `for`-Schleife kann man doch mehrere Ausdrücke durch Komma getrennt angeben.

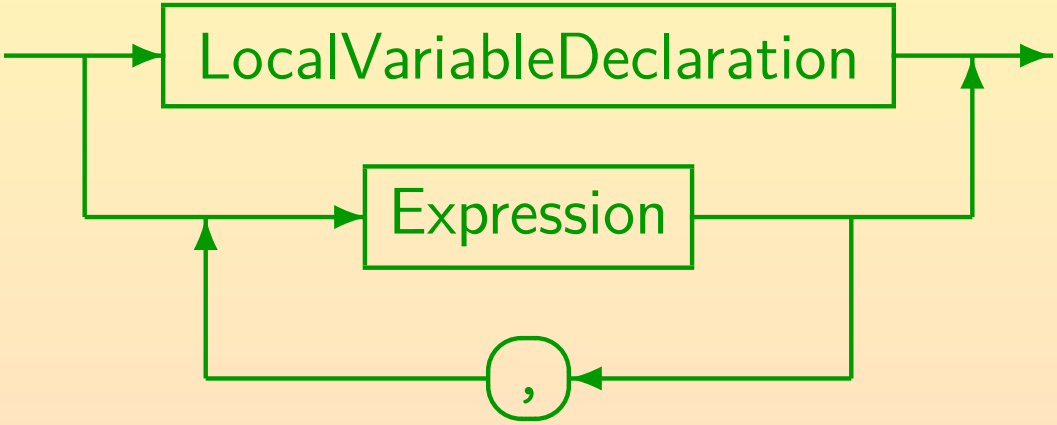
# For-Schleife (9)

Syntaxdiagramm (ForStatement):

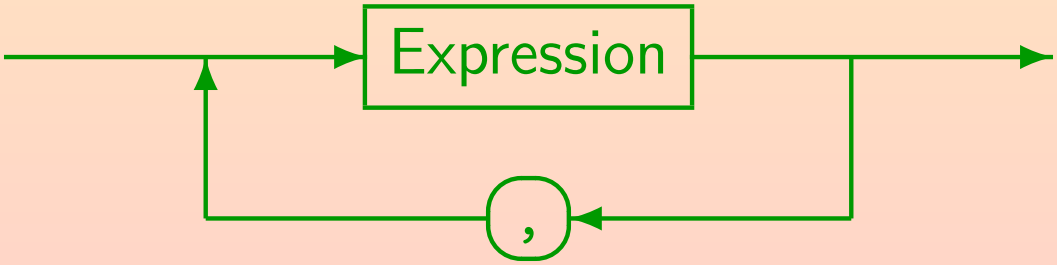


# For-Schleife (10)

ForInit:




ForUpdate:




Als Expression sind nur die Typen erlaubt, die auch als Statement verwendet werden können: Zuweisungen, Pre-/Post-Increment/Decrement und Methodenaufrufe.

# For-Schleife (11)

Aufgabe:

- Was gibt dieses Programmstück aus?

```
System.out.print("Bitte n eingeben: ");
int n = scanner.nextInt();
for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j++)
                System.out.print('*');
        System.out.println();
}
```

# "Foreach"-Schleife (1)

*

- Seit Java 5 gibt es noch eine syntaktische Variante der `for`-Schleife zum Durchlaufen von Datenstrukturen, über die man iterieren kann (Arrays, Listen, etc.).

  Es muss sich um Arrays handeln oder Typen, die das Interface "`Iterable`" implementieren. Dies wird später im Kapitel über Collection-Typen besprochen.

- Diese Schleife wird auch "foreach"-Schleife genannt, verwendet aber das Schlüsselwort "`for`".

  In Pseudocode schreibt man üblicherweise "`foreach`" für solche Schleifen. In der Java-Spezifikation heisst sie "Enhanced For Loop".

- Diese Schleife ist aber wieder nur eine Abkürzung, sie bringt keine grundsätzlich neuen Möglichkeiten.

# "Foreach"-Schleife (2)

- In dieser Schleife deklariert man eine Variable, die über den Elementen der Datenstruktur läuft.
- Wenn `a` z.B. den Typ `double[]` hat, kann man es mit folgender Schleife ausgeben:

```
for(double x : a)
        System.out.println(x);
```

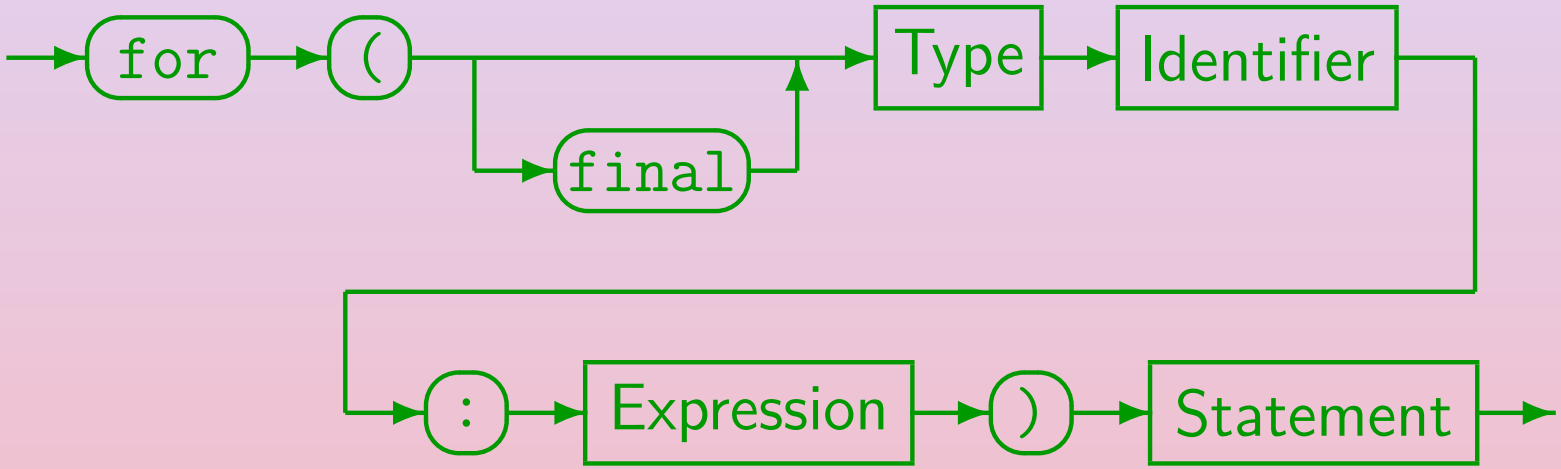
- Dies ist eine Abkürzung für (mit neuer Variablen `i`):

```
for(int i = 0; i < a.length; i++) {
        double x = a[i];
        System.out.println(x);
}
```

Wenn man für a einen komplizierteren Ausdruck schreibt, wird der nur einmal vor Beginn der Schleife ausgewertet und in einer Hilfsvariablen gespeichert.

# "Foreach"-Schleife (3)

- Syntaxgraph (EnhancedForStatement):




Falls `C` einen Typ hat, der Untertyp von `Iterable` ist, so steht

```
for(T X : C) S;
```

für

```
for(I i = C.iterator(); i.hasNext(); ) {
        T X = i.next();
        S;
}
```

Dabei sei `I` der Ergebnistyp von `C.iterator()` und `i` ein neuer Variablenname.

# Inhalt


1. Allgemeines
2. Blöcke
3. Bedingte Anweisungen
4. Schleifen
5. Sprünge

# Break Statement (1)

- Die Anweisung "`break;`" beendet die Schleife oder den Switch, in dem es sich befindet.

  Es funktioniert für alle drei Schleifentypen: `while`, `do`, `for`. Falls mehrere Schleifen (oder Switches) geschachtelt sind, wird immer nur die innerste Schleife (bzw. Switch) beendet, in der sich das `break` befindet. Es gibt aber eine Variante mit "Label" (Marke), die auch andere Anweisungen beenden kann (s.u.).

- Auf diese Art kann eine Schleife nicht nur über die Bedingung im Kopf beendet werden, sondern man kann auch an beliebiger Stelle im Rumpf entscheiden, sie zu verlassen.

  Das ist manchmal sehr praktisch. Auf der anderen Seite macht es die Programme unübersichtlicher.

# Break Statement (2)

- Beispiel: Angegenommen, man möchte die Position des ersten Nicht-Leerzeichens in einem `String s` ermitteln:

```
int i;
for(i = 0; i < s.length(); i++) {
        if(s.charAt(i) != ' ')
                break;
}
if(i < s.length())
        System.out.println("Position: " + i);
else
        System.out.println("Nur Leerzeichen!");
```

Die Schleife läuft im Prinzip über dem ganzen String, wird aber abgebrochen, sobald man ein Leerzeichen findet. Da man die Laufvariable vor der Schleife deklariert hat, kann man nach der Schleife noch auf sie zugreifen. Ist `i` noch ein gültiger Index, wurde die Schleife mit `break` beendet (Nicht-LZ gefunden).

# Break Statement (3)

- Man kann diese Schleife aber auch ohne `break` schreiben:

```
int i = 0;
while(i < s.length() && s.charAt(i) == ' ')
        i++;
if(i < s.length())
        System.out.println("Position: " + i);
else
        System.out.println("Nur Leerzeichen!");
```

  Wenn man hier `for` verwendet, bekommt man eine Schleife mit leerem Rumpf: "`for(...) ;`" Die ganze Arbeit geschieht schon in der Schleifenkontrolle.

- Ich würde diese Schleife vorziehen.

  Das ist aber wohl eine Geschmacksfrage. Ich verwende `break`, wenn im Schleifenrumpf erst eine komplexere Berechnung nötig ist, bevor man ggf. die Schleife abbrechen kann.

# Break Statement (4)

- Bisher hatte jede Anweisung nur einen Eingang und einen Ausgang.
- Mit dem `break`-Statement kommt jetzt eine weitere Möglichkeit für einen Ausgang hinzu.
- Bisher konnte man bei einer sequentiellen Komposition von Anweisungen ("Block") sicher sein, dass sie auch wirklich alle ausgeführt werden.

  Von Endlosschleifen abgesehen.

- Jetzt muss man sich die Anweisungen genauer anschauen, ob sie vielleicht ein "`break;`" enthalten.

  Hinweis: Exceptions (siehe Kapitel 9) können auch dazu führen, dass sogar mitten in einem Ausdruck die Auswertung plötzlich endet. Sie wird dann ggf. in einer "`catch`"-Klausel weiter außen fortgesetzt.

# Break Statement (5)

- Java hat (im Gegensatz zu C++) noch eine Form des `break`-Statements mit dem man auch andere Anweisungen als die innerste Schleife / den innersten Switch verlassen kann.

  C++ hat eine allgemeine `goto`-Anweisung, mit der man zu (fast) beliebigen Punkten in der Methode springen kann. Die Verwendung von `goto` gilt jedoch als sehr schlechter Stil. Es gibt z.B. einen häufig zitierten Artikel "Go To Statement Considered Harmful" von Edsger W. Dijkstra (in den Communications of the ACM, Vol. 11, No. 3, März 1968, S. 147–148).
  Er beginnt mit den Worten: "For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce."
  Java hat kein `goto`. Es ist zwar ein reserviertes Wort, aber nur, um bessere Fehlermeldungen für Programmierer zu erzeugen, die von C/C++ kommen.
  Die meisten Fälle, bei denen ein `goto` eventuell nützlich sein könnte, sind in Java durch die erweiterte `break`-Anweisung abgedeckt.

# Break Statement (6)

- Für die erweiterte Form der `break`-Anweisung markiert man ein Statement *S* mit einem "Label" (einer Marke):

  `XYZ: S`

  Dabei kann *S* komplex und tief geschachtelt sein.

  Es muss auch keine Schleife und kein Switch sein, obwohl dies sicher die häufigsten Fälle sind. Da Label leicht am folgenden "`:`" zu erkennen sind, gibt es keine Namenskonflikte mit anderen Bezeichnern. Man kann z.B. den gleichen Bezeichner als Label und als lokale Variable verwenden.

- Irgendwo im Innern von "*S*" schreibt man dann:

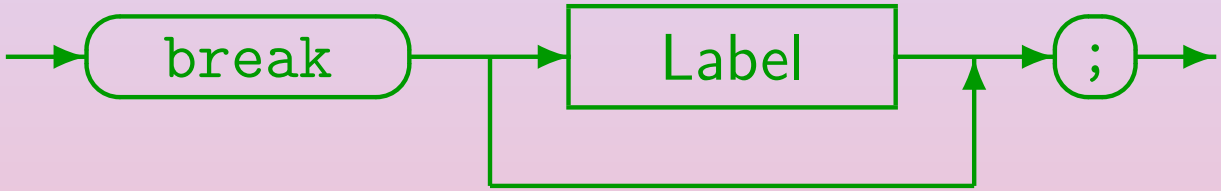
  `break XYZ;`

  Die `break`-Anweisung muss im Innern einer Anweisung stehen, die mit dem entsprechenden Label markiert ist (keine beliebigen Sprungziele).

- Es wird dann die Anweisung "*S*" beendet, und die Ausführung mit der folgenden Anweisung fortgesetzt.

# Break Statement (7)

- Syntax-Diagramm (BreakStatement):




Es ist ein Syntaxfehler, `break` ohne Label außerhalb von `while`, `do`, `for`, `switch` zu verwenden. Es muss aber nicht direkt in diesen Statements stehen. Es ist z.B. ganz typisch, dass es in einem `if`-Statement steht, das seinerseits im Innern einer Schleife ist. Mit Label geht es in einem beliebigen Statement, das mit diesem Label markiert ist.

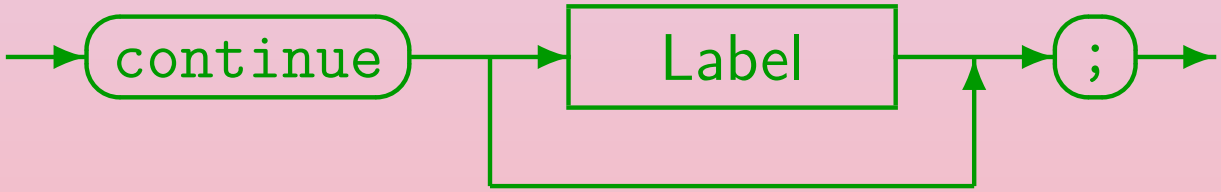
- Syntax-Diagramm (LabeledStatement):

# Continue Statement

- Mit `continue;` springt man zum Ende des Schleifenrumpfes, d.h. es beginnt sofort der nächste Schleifendurchlauf.

  Bei der `for`-Schleife wird der Schritt-Ausdruck noch ausgewertet, d.h. die Laufvariable wird weitergeschaltet.

- Syntax-Diagramm (continue-Statement):




  Es ist ein Syntaxfehler, `continue` außerhalb von `while`, `do`, `for` zu verwenden.
  Auch bei der Variante mit Label muss der Label vor einer Schleife stehen.
  Im Gegensatz zu `break` hat `continue` keine Beziehung zum `switch`-Statement.

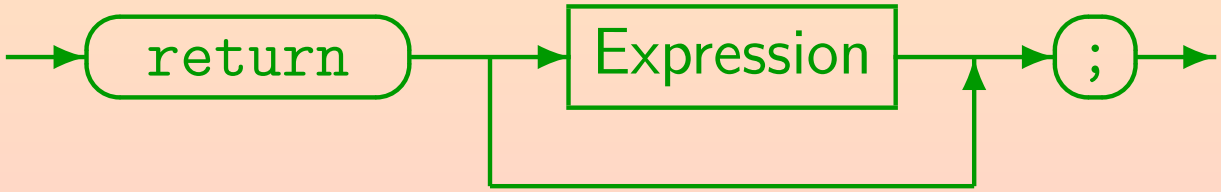
- Ich verwende `continue` äußerst selten (bisher vermutlich nie).

# Return Statement

- Mit dem Return-Statement wird die aktuelle Methode verlassen und ggf. ein Rückgabewert festgelegt.

  In gewisser Weise findet hier auch ein Sprung statt: Man geht direkt zum Ende der Methode. Pascal hat kein Return-Statement. Das Return-Statement ist allerdings nützlich und allgemein akzeptiert.

- Syntax-Diagramm (Return-Statement):




- Methoden werden in Kapitel 11 ausführlich besprochen.

# Statements: Übersicht (1)

- Zuweisung: `i = 1;`
- Inkrement/Dekrement: `i++;`
- Methoden-Aufruf: `obj.method(...);`

  Bei statischer Methode (Klassen-Methode) Klassen-Name statt Objekt.
- Deklaration, ggf. mit Initialisierung: `int i = 0;`
- Block: `{ i = 1; j = i; ...}`
- `if` mit `else`: `if(i > 0)` $S_1$ `else` $S_2$

  Dabei können $S_1$ und $S_2$ fast beliebige Statements sein, z.B. einzelne Zuweisungen oder ganze Blöcke. Eine einzelne Deklaration ist aber verboten (im Block natürlich nicht). Außerdem kann $S_1$ nicht ein `if` ohne `else` sein.
- `if` ohne `else`: `if(i > 0)` $S$

# Statements: Übersicht (2)

- Switch: `switch(i) { case 1: m = "Jan"; break; ...}`
- `while`-Schleife: `while(i < 100)` $S$
- `do`-Schleife: `do` $S$ `while(i < 100);`
- `for`-Schleife: `for(int i = 0; i < 100; i++)` $S$
- "foreach"-Schleife: `for(Elem e : Datenstruktur)` $S$
- `break`-Anweisung (beendet Schleife oder Switch): `break;`

  Es gibt noch eine Variante mit Label.
- Sprung zum nächsten Schleifendurchlauf: `continue;`
- `return`-Anweisung (beendet Methode): `return i;`

  Bei `void`-Methoden nur `return;`

# Statements: Übersicht (3)

- Exception auslösen:
  `throw new IndexOutOfBoundsException();`
- Exception behandeln:
  `try {...} catch(IOException e){...} finally {...}`

  Es kann mehrere `catch`-Klauseln geben. Die `finally`-Klausel ist optional.

  Es muss aber mindestens eine `catch`-Klausel oder die `finally`-Klausel geben.
- Parallele Zugriffe durch Sperren verhindern:
  `synchronized(obj) {...}`
- Zusicherung: `assert i > 0 : "i muss positiv sein";`

  Den Doppelpunkt und die Meldung kann man auch weglassen.
- Leeres Statement: `;`
- Labeled Statement: `HauptSchleife: while(...) {...}`