Einheit 2

Syntax und Semantik
Einfache Programme
Verzweigungen
Schleifen

© Hanspeter Mössenböck, Herbert Prähofer

Syntax und Semantik von Programmiersprachen

© Hanspeter Mössenböck, Herbert Prähofer

Beschreibung von Programmiersprachen

Syntax

Regeln, nach denen Sätze gebaut werden dürfen

z.B.: Zuweisung = Variable "←" Ausdruck.

Semantik

Bedeutung der Sätze

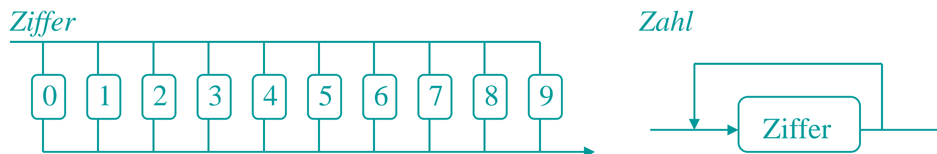
z.B.: werte Ausdruck aus und weise ihn der Variablen zu

Grammatik

Menge von Syntaxregeln

z.B. Grammatik der ganzen Zahlen

Ziffer = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
Zahl = Ziffer {Ziffer}.



EBNF (Erweiterte Backus-Naur-Form)

Metazeichen	Bedeutung	Beispiel	beschreibt
=	trennt Regelseiten		
.	schließt Regel ab		
|	trennt Alternativen	x | y	x, y
( )	klammert Alternativen	(x | y) z	xz, yz
[ ]	wahlweises Vorkommen	[x] y	xy, y
{ }	0..n-maliges Vorkommen	{x} y	y, xy, xxy, xxxy, ...

Beispiele

Grammatik der Gleitkommazahlen

Zahl = Ziffer {Ziffer}.
Gleitkommazahl = Zahl "." Zahl ["E" ["+" | "-"] Zahl].

Syntax und Semantik von Java

Beispiel

Grammatik der If-Anweisung

IfAnweisung = "if" "(" Ausdruck ")" Anweisung ["else" Anweisung].

Compiler und Syntaxfehler

- Compiler liest ein Java-Programm und überprüft es auf *syntaktischische* Korrektheit
- Ist das Programm nicht korrekt, wird ein Syntaxfehler angezeigt, d.h.
  - eine Fehlermeldung ausgegeben und
  - das Programm nicht übersetzt.

```
C:\Daten\AktuelleDaten\Rottenmann\SW1_VL_2003W\JavaPgrams\Unit_01\GGTProg
ram.java:12: ';' expected

                          x = 5
                               ^
1 error
Compilierung beendet
```

- Fehlermeldungen zeigen den Fehler und die Position des Erkennens des Fehlers an
  - Achtung: Fehlermeldungen können oft auch verwirrend sein !!!

Syntaxfehler Hinweise

- In der Fehlermeldung ist die genaue Stelle markiert, wo ein Fehler ***erkannt*** wurde; dies ist nicht immer die Stelle, wo der Fehler ***gemacht*** wurde.
- Immer zuerst auf den ersten Syntaxfehler schauen; es treten oft eine Reihe von Folgefehler auf.
- Schwierig zu findende Fehler sind Klammerfehler (Klammer vergessen); Daher schon beim Schreiben auf die Klammerung achten.

© Hanspeter Mössenböck, Herbert Prähofer

Typische Syntaxfehlermeldungen (1)

- Strichpunkt vergessen
  - **`´;´ expected`**
- Klassenname != Dateiname
  (z.B. Klasse `dual` klein und Datei `Dual.java` groß)
  - **`class dual is public, should be declared in a file named dual.java`**
- Variable oder Methode nicht bekannt
  (z.B. Deklaration vergessen oder Variable, Methode falsch geschrieben)
  - **`cannot resolve symbol symbol  : variable ch`**
  - **`cannot resolve symbol symbol  : method pute (java.lang.String)`**
- Zeichenkette nicht geschlossen
  (z.B. `TextIO.put("Bitte Ziffernfolge eingeben: );`)
  - **`unclosed string literal TextIO.put("Bitte Ziffernfolge eingeben: );`**
- runde Klammer nicht geschlossen
  (z.B. `TextIO.put("Bitte Ziffernfolge eingeben: ";)`
  - **`')' expected TextIO.put("Bitte Ziffernfolge eingeben: ";`**
- Geschwungene Klammer-Zu fehlt:
  - **`'}' expected`** (ganz am Ende des Programms)
  - **`Illigal start of expression`**

© Hanspeter Mössenböck, Herbert Prähofer

Typische Syntaxfehlermeldungen(2)

- Geschwungene Klammer-Auf fehlt:
  - leider oft sehr unverständliche Fehlermeldungen, weil damit ein Programmteil zu früh zu Ende ist und ein anderer beginnen soll; ein Folgefehler ist normalerweise

    `'class' or 'interface' expected`
- Doppelte Deklaration einer Variablen (z.B. `int n; int n = 2;`)
  - **`n is already defined in main(java.lang.String[] int n);`**
- Variable bei der Verwendung nicht initialisiert
  - **`variable n might not have been initialized  n = n+1;`**
- Zuweisung zwischen inkompatiblen Typen (z.B. **`int i = true;`**)
  - **`incompatible types found : int required: int i = true;`**
- Zuweisung zwischen incompatiblen Zahlen (z.B. `int x = 1.1`)
  - **`possible loss of precision found : double  required: int  int x = 1.1`**
- In Bedingung = statt == verwendet (z.B. `if (ch = '1') {}`)
  - **`incompatible types found  : char  required: boolean if (ch = '1') {`**

Sequentielle Programme

Grundsymbole

Namen

bezeichnen Variablen, Typen, ... in einem Programm
- bestehen aus Buchstaben, Ziffern und "_"
- beginnen mit Buchstaben
- beliebig lang
- Groß-/Kleinschreibung signifikant

```
x
x17
myVar
my_Var
```

Schlüsselwörter

- heben Programmeile hervor
- dürfen nicht als Namen verwendet werden

```
if
while
```

Zahlen

- ganze Zahlen (dezimal oder hexadezimal)
- Gleitkommazahlen

376	dezimal
0x1A5	$1*16^2+10*16^1+5*16^0$
3.14	Gleitkommazahl

Zeichketten

- beliebige Zeichen zwischen Hochkommas
- dürfen nicht über Zeilengrenzen gehen
- " wird als \" geschrieben

```
"a simple string"
"sie sagte \"Hallo\""
```

Variablendeklarationen

Jede Variable muss vor ihrer Verwendung deklariert werden

- macht den Namen und den Typ der Variablen bekannt
- Compiler reserviert Speicherplatz für die Variable

int x;	deklariert eine Variable x vom Typ int (integer)
short a, b;	deklariert 2 Variablen a und b vom Typ short (short integer)

Ganzzahlige Typen

byte	8-Bit-Zahl	$-2^7 .. 2^7-1$	(-128 .. 127)
short	16-Bit-Zahl	$-2^{15} .. 2^{15}-1$	(-32768 .. 32767)
int	32-Bit-Zahl	$-2^{31} .. 2^{31}-1$	(-2 147 483 648 .. 2 147 483 647)
long	64-Bit-Zahl	$-2^{63} .. 2^{63}-1$	

Initialisierungen

int x = 100;	deklariert int-Variable x; weist ihr den Anfangswert 100 zu
short a = 0, b = 1;	deklariert 2 short-Variablen a und b mit Anfangswerten

Kommentare

Geben Erläuterungen zum Programm

Zeilenendekommentare

- beginnen mit //
- gehen bis zum Zeilenende

```
int sum;  // total sales
```

Klammerkommentare

- durch /* ... */ begrenzt
- können über mehrere Zeilen gehen
- dürfen nicht geschachtelt werden
- zum "Auskommentieren" von Programmteilen

```
/* Das ist ein längerer
   Kommentar, der über
   mehrere Zeilen geht */
```

Sinnvoll kommentieren!

- alles kommentieren, was Erklärung bedarf
- statt unklares Programm mit Kommentar, besser klares Programm ohne Kommentar
- nicht kommentieren, was ohnehin schon im Programm steht; folgendes ist z.B. unsinnig
  int sum; // Summe

Sprache in Kommentaren und Namen

Deutsch

+ einfacher

Englisch

+ meist kürzer
+ paßt besser zu den englischen Schlüsselwörtern (if, while, ...)
+ Programm kann international verteilt werden (z.B. über das Web)

Jedenfalls: Deutsch und Englisch nicht mischen!!

Namenswahl für Variablen

Einige Tipps

- lesbar aber nicht zu lang
  z.B. *sum*, *value*
- Hilfsvariablen, die man nur über kurze Strecken braucht, eher kurz:
  z.B. *i*, *j*, *x*
- Variablen, die man im ganzen Programm braucht, eher länger:
  z.B. *inputText*
- mit Kleinbuchstaben beginnen,
  Worttrennung durch Großbuchstaben
  z.B. *inputText*
- Englisch oder Deutsch?
- Konstanten immer groß und mit "_" getrennt, z.B. MAX_VALUE

© Hanspeter Mössenböck, Herbert Prähofer

Konstantendeklarationen

Initialisierte "Variablen", deren Wert man nicht mehr ändern kann

```
static final int MAX = 100;
```

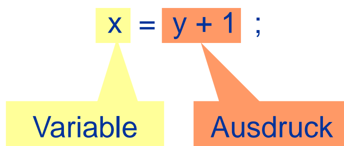
Zweck

- bessere Lesbarkeit (MAX ist lesbarer als 100)
- bessere Wartbarkeit (Wert muß nur an 1 Stelle geändert werden)

Konstantendeklaration muß auf Klassenebene stehen (s. später)

© Hanspeter Mössenböck, Herbert Prähofer

Zuweisungen



1. berechne den Ausdruck
2. speichere seinen Wert in der Variablen

Bedingung: linke und rechte Seite müssen zuweisungskompatibel sein
- müssen dieselben Typen haben, oder
- Typ $_{links} \supseteq$ Typ $_{rechts}$

Hierarchie der ganzzahligen Typen

long ⊇ int ⊇ short ⊇ byte

Beispiele

```
int i, j; short s; byte b;
i = j;   // ok: derselbe Typ
i = 300; // ok (Zahlkonstanten sind int)
b = 300; // falsch: 300 paßt nicht in byte
i = s; // ok
s = i; // falsch
```

Statische Typenprüfung: Compiler prüft:
- daß Variablen nur erlaubte Werte enthalten
- daß auf Werte nur erlaubte Operationen ausgeführt werden

Arithmetische Ausdrücke

Vereinfachte Grammatik

Expr = Operand {BinaryOperator Operand}.
Operand = [UnaryOperator] (identifier | number | "(" Expr ")").

Binäre Operatoren

+	Addition				
-	Subtraktion				
*	Multiplikation				
/	Division, Ergebnis ganzzahlig	4/3 = 1	(-4)/3 = -1	4/(-3) = -1	(-4)/(-3) = 1
%	Modulo (Divisionsrest)	4%3 = 1	(-4)%3 = -1	4%(-3) = 1	(-4)/(-3) = -1

Unäre Operatoren

+	Identität (+x = x)
-	Vorzeichenumkehr

Typregeln in arithm. Ausdrücken

Vorrangregeln

- Punktrechnung (*, /, %) vor Strichrechnung (+, -)
- Unäre Operatoren binden stärker als binäre
  z.B.: 3 + 4 * -2 ergibt -5

Typregeln

Operandentypen	byte, short, int, long
Ergebnistyp	- wenn mindestens 1 Operand long ist ⇒ long - sonst ⇒ int

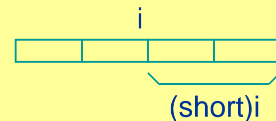
Beispiele

```
short s; int i; long x;
x = x + i;          // long
i = s + 1;          // int (1 ist vom Typ int)
s = (short)(s + 1);
                    // Typumwandlung nötig
```

Typumwandlung (type cast)

(*type*)*expression*

- wandelt Typ von *expression* in *type* um
- dabei kann etwas abgeschnitten werden



Increment und Decrement

Variablenzugriff kombiniert mit Addition/Subtraktion

x++	nimmt den Wert von x und erhöht x anschließend um 1
++x	erhöht x um 1 und nimmt anschließend den erhöhten Wert
x--	nimmt den Wert von x und erniedrigt x anschließend um 1
--x	erniedrigt x um 1 und nimmt anschließend den erniedrigten Wert

Beispiele

```
x = 1;  y = x++ * 3;     // x = 2, y = 3
x = 1;  y = ++x * 3;     // x = 2, y = 6
```

Darf nur auf Variablen angewendet werden (nicht auf Ausdrücke)

```
y = (x + 1)++;           // falsch!
```

Kann auch als eigenständige Anweisung verwendet werden

```
x = 1; x++;              // x = 2
```

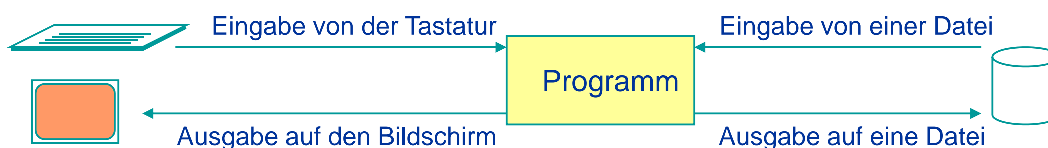
Zuweisungsoperatoren

Arithmetischen Operationen lassen sich mit Zuweisung kombinieren

	Kurzform	*Langform*
+=	x += y;	x = x + y;
-=	x -= y;	x = x - y;
*=	x *= y;	x = x * y;
/=	x /= y;	x = x / y;
%=	x %= y;	x = x % y;

Spart Schreibarbeit, ist aber nicht schneller als die Langform

Eingabe und Ausgabe von Werten



Eingabe

```
int x = In.readInt();   // liest eine Zahl vom Eingabestrom
if (In.done()) ...      // liefert true oder false, je nachdem, ob Lesen erfolgreich war
In.open("MyFile.txt");  // öffnet Datei als neuen Eingabestrom
In.close();             // schließt Datei und kehrt zum alten Eingabestrom zurück
```

Ausgabe

```
Out.print(x);            // gibt x auf dem Ausgabestrom aus (x kann von bel. Typ sein)
Out.println(x);          // gibt x aus und beginnt eine neue Zeile
Out.open("MyFile.txt");  // öffnet Datei als neuen Ausgabestrom
Out.close();             // schließt Datei und kehrt zum alten Ausgabestrom zurück
```

http://www.ssw.uni-lnz.ac.at/Misc/JavaBuch

Besonderheiten zur Eingabe

Eingabe von Tastatur

Eintippen von: 12 100

füllt Lesepuffer

Programm:

```
int x = In.readInt();   // liest 12
int y = In.readInt();   // liest 100
int z = In.readInt();   // blockiert, bis Lesepuffer wieder gefüllt ist
```

Ende der Eingabe: Eingabe von Strg-Z in leere Zeile

Eingabe von Datei

kein Lesepuffer, *In.readInt()* liest direkt von der Datei

Ende der Eingabe wird automatisch erkannt (kein Strg-Z nötig)

Grundstruktur von Java-Programmen

```
class ProgramName {

  public static void main (String[] arg) {
    ... // Deklarationen
    ... // Anweisungen
  }

}
```

Text muß in einer Datei namens *ProgramName*.java stehen

Beispiel

```
class Sample {
  public static void main (String[] arg) {
    Out.print("Geben Sie 2 Zahlen ein:");
    int a = In.readInt();
    int b = In.readInt();
    Out.println("Summe = " + (a + b));
  }
}
```

Text steht in Datei Sample.java

Beispielprogramm: Arithmetische Operationen

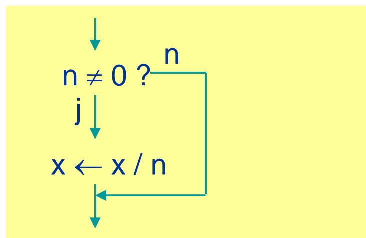
- Beispiel: Berechnung von Volumen, Umfang, Oberfläche eines Quaders
  - Einlesen von Breite, Länge, Höhe
  - Berechnung der Volumen, Umfang, Oberfläche
  - Ausgabe der Ergebnisse

```
public class Quader {

    public static void main(String[] args) {

        int width, length, height;
        int surface, volume, surrounding;

        // Printing a header
        Out.println();
        Out.println("Programm zur Berechnung von Oberfläche, Umfang und Volumen");
        Out.println("   eines beliebigen Quaders ");
        Out.println("==========================================================");
        Out.println();

        // Reading width, length, and height
        Out.print("Bitte Breite eingeben [m]: ");
        width = In.readInt();

        …
```

…. Übungsarbeit

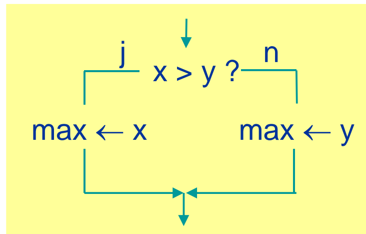
Verzweigungen

If-Anweisung




```
if (n != 0)
    x = x / n;
```

ohne else-Zweig




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

mit else-Zweig

Syntax

IfStatement = "**if**" "**(**" Expression "**)**" Statement ["**else**" Statement].

Anweisungsblöcke

Wenn then-Zweig oder else-Zweig aus mehr als 1 Anweisung bestehen, müssen sie durch { ... } geklammert werden.

Statement = Assignment | IfStatement | ... | **Block**;
Block = "**{**" {Statement} "**}**".

Beispiel:

```
if (x < 0) {
  negNumbers++;
  Out.print(-x);
} else {
  posNumbers++;
  Out.print(x);
}
```

Einrückungen

- erhöhen die Lesbarkeit (machen Programmstruktur besser sichtbar)
- Einrückungstiefe: 1 Tabulator oder 2 oder 4 Leerzeichen

```
if (n != 0)
    x = x / n;
```

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

```
if (x < 0) {
    negNumbers++; Out.print(-x);
} else {
    posNumbers++; Out.print(x);
}
```

Kurze If-Anweisungen können auch in *einer* Zeile geschrieben werden

```
if (n != 0) x = x / n;
if (x > y) max = x;
else max = y;
```

Dangling Else

```
if (a > b)
  if (a != 0) max = a;
else
  max = b;
```

Mehrdeutigkeit! Zu welchem if gehört das else?

Regel: else gehört immer zum unmittelbar vorausgegangenen if.

Wenn man das nicht will, muß man die Anweisung so schreiben:

```
if (a > b) {
  if (a != 0) max = a;
} else {
  max = b;
}
```

Wir wollen immer Klammern verwenden!!

Vergleichsoperatoren

Vergleich zweier Werte liefert wahr (*true*) oder falsch (*false*)

	Bedeutung	*Beispiel*
==	gleich	x == 3
!=	ungleich	x != y
>	größer	4 > 3
<	kleiner	x+1 < 0
>=	größer oder gleich	x <= y
<=	kleiner oder gleich	x >= y

Wird z.B. in If-Anweisung verwendet

```
if (x == 0) Out.println("x is zero");
```

Achtung: "=" ist in Java kein Vergleich, sondern eine Zuweisung

```
if (x = 0) Out.println("x is zero");   // Compiler meldet einen Fehler!
```

© Hanspeter Mössenböck, Herbert Prähofer

Zusammengesetzte Vergleiche

&& Und-Verknüpfung

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

|| Oder-Verknüpfung

x	y	x || y
true	true	true
true	false	true
false	true	true
false	false	false

! Nicht-Verknüpfung

x	!x
true	false
false	true

Beispiel

```
if (x >= 0 && x <= 10 || x >= 100 && x <= 110) y = x;
```

Vorrangregeln

! bindet stärker als &&
&& bindet stärker als ||

Vorrangregeln können durch Klammerung umgangen werden:

```
if (x > 0 && (y == 1 || y == 7)) ...
```

© Hanspeter Mössenböck, Herbert Prähofer

Kurzschlußauswertung

Zusammengesetzter Vergleich wird abgebrochen, sobald Ergebnis feststeh

```
if ( a != 0  && b / a > 0 ) x = 0;
```

wenn false, ist gesamter Ausdruck false

äquivalent zu

```
if (a != 0)
  if (b / a > 0) x = 0;
```

```
if ( a == 0  || b / a > 0 ) x = 1;
```

wenn true, ist gesamter Ausdruck true

```
if (a == 0)
  x = 1;
else if (b / a > 0)
  x = 1;
```

Datentyp boolean

nach George Boole: Mathematiker, 1815-1864

Datentyp wie int mit den beiden Werten *true* und *false*

Beispiele

```
boolean p, q;
p = false;
q = x > 0;
p = p || q && x < 10;
```

Beachte

- Jeder Vergleich liefert einen Wert vom Typ boolean
- Boolesche Werte können mit &&, || und ! verknüpft werden
- Boolesche Werte können in boolean-Variablen abgespeichert werden ("flags")
- Namen für boolean-Variablen sollten mit Adjektiv beginnen: equal, full

Programmvergleich

Welches der beiden Programme ist besser?

```
if (a > b)
    if (a > c)
        max = a;
    else
        max = c;
else
    if (b > c)
        max = b;
    else
        max = c;
```

```
max = a;
if (b > max) max = b;
if (c > max) max = c;
```

Was heißt "besser"?

- Kürze: das 2. Programm ist kürzer
- Effizienz:
  - 1. Programm braucht immer 2 Vergleiche und 1 Zuweisung
  - 2. Programm braucht immer 2 Vergleiche und im Schnitt 2 Zuweisungen
- Lesbarkeit?

Negation zusammengesetzter Ausdrücke

Regeln von DeMorgan

```
! (a && b)  ⇒  ! a || ! b
! (a || b)  ⇒  ! a && ! b
```

Diese Regeln helfen beim Bilden von Assertionen

```
if (x >= 0 && x < 10) {
  ...
} else { // x < 0 || x >= 10
  ...
}
```

If-Kaskade

- Geschachtelte If-Anweisung
  - im else-Zweig steht wieder eine if-Anweisung

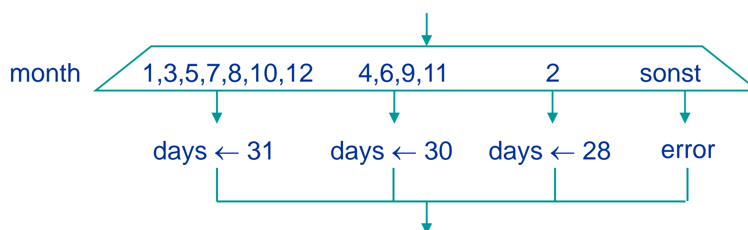
```
if (ch>='a' && ch<='c') {
    println("a...c");
}
else {
    if  (ch>'c' && ch<='k') {
        println("d...k");
    }
    else {
        if (ch>'k' && ch<='o') {
                println("l...o");
        }
        else {
                println("p...z");
        }
    }
}
```

- Besser If-Kaskade
  - Vermeiden { } bei if im else-Zweig
  - Keine Einrückung bei else

```
if (ch>='a' && ch<='c') {
    println("a...c");
} else if  (ch>'c' && ch<='k') {
    println("d...k");
} else if  (ch>'k' && ch<='o') {
    println("l...o");
} else {
    println("p...z");
}
```

Switch-Anweisung

Mehrwegverzweigung



In Java

```
switch (month) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    days = 31; break;
  case 4: case 6: case 9: case 11:
    days= 30; break;
  case 2:
    days = 28; break;
  default:
    Out.println("error");
}
```

Semantik der Switch-Anweisung

Switch-Ausdruck

```
switch (month) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    days = 31; break;
  case 4: case 6: case 9: case 11:
    days= 30; break;
  case 2:
    days = 28; break;
  default:
    Out.println("error");
}
```

Break-Anweisung

- springt ans Ende der Switch-Anweisung
- wenn *break* fehlt, läuft Programm über nächste case-Marke weiter (häufige Fehlerursache!!)

Semantik

1. berechne Switch-Ausdruck
2. springe zur passenden case-Marke
   - wenn keine paßt, springe zu default
   - wenn kein default angegeben, springe ans Ende der Switch-Anweisung

Bedingungen

1. case-Marken müssen Konstanten sein
2. ihr Typ muß zu Typ des switch-Ausdrucks passen
3. case-Marken müssen voneinander verschieden sein

Syntax der Switch-Anweisung

Statement	= Assignment | IfStatement | SwitchStatement | ... | Block.
SwitchStatement	= "**switch**" "**(**" Expression "**)**" "**{**" {LabelSeq StatementSeq} "**}**".
LabelSeq	= Label {Label}.
StatementSeq	= Statement {Statement}.
Label	= "**case**" ConstantExpression "**:**" | "**default**" "**:**".

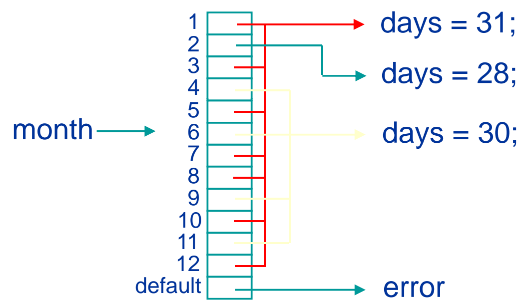
Unterschied zwischen If und Switch

```
if (month==1 || month==3 || month==5
    || month==7 || month==8 || month==10
    || month==12)
    days = 31;
else if (month==4 || month==6
    || month==9 || month==11)
    days = 30;
else if (month==2)
    days = 28;
else Out.println("error");
```

```
switch (month) {
    case 1: case 3: case 5: case 7:
    case 8: case 10: case 12:
        days = 31; break;
    case 4: case 6: case 9: case 11:
        days= 30; break;
    case 2:
        days = 28; break;
    default:
        Out.println("error");
}
```

prüft Bedingungen sequentiell

benutzt Sprungtabelle



© Hanspeter Mössenböck, Herbert Prähofer