

Einheit 2

Syntax und Semantik
Einfache Programme
Verzweigungen
Schleifen



Syntax und Semantik von Programmiersprachen



Beschreibung von Programmiersprachen

Syntax

Regeln, nach denen Sätze gebaut werden dürfen

z.B.: Zuweisung = Variable " \leftarrow " Ausdruck.

Semantik

Bedeutung der Sätze

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

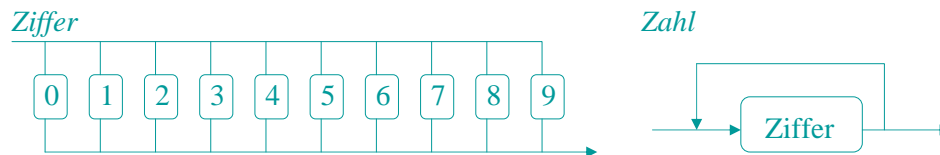
Grammatik

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	$\{x\} y$	$y, xy, xxy, xxxy, \dots$

Beispiele

Grammatik der Gleitkommazahlen

Zahl = Ziffer {Ziffer}.

Gleitkommazahl = Zahl "." Zahl ["E" ["+" | "-"] Zahl].



Beispiel

Grammatik der If-Anweisung

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



Compiler und Syntaxfehler

- Compiler liest ein Java-Programm und überprüft es auf *syntaktische* 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\GGTProgram.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.



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`



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 inkompatiblen 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*162+10*161+5*160
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
```



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)



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!!



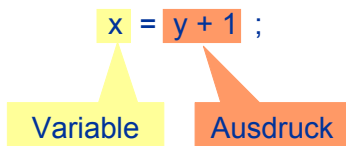
Namenswahl für Variablen und Konstanten

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`



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
- $\text{Typ}_{\text{links}} \supseteq \text{Typ}_{\text{rechts}}$

Hierarchie der ganzzahligen Typen

`long \supseteq int \supseteq short \supseteq 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 \Rightarrow long
 - sonst \Rightarrow int

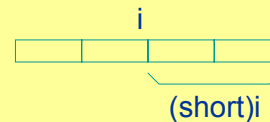
Beispiele

```
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
```



Multiplikation/Division mit Zweierpotenzen

Mit Shift-Operationen effizient implementierbar

<i>Multiplikation</i>		<i>Division</i>		
$x * 2$	$x \ll 1$	$x / 2$	$x \gg 1$	Division nur bei positiven Zahlen durch Shift ersetzbar
$x * 4$	$x \ll 2$	$x / 4$	$x \gg 2$	
$x * 8$	$x \ll 3$	$x / 8$	$x \gg 3$	
$x * 16$	$x \ll 4$	$x / 16$	$x \gg 4$	
...	

Beispiele

$x = 3;$

0000 0011

$x = x \ll 2; // 12$

0000 1100

$x = -3;$

1111 1101

$x = x \ll 1; // -6$

1111 1010

$x = 15;$

0000 1111

$x = x \gg 2; // 3$

0000 0011



Zuweisungsoperatoren

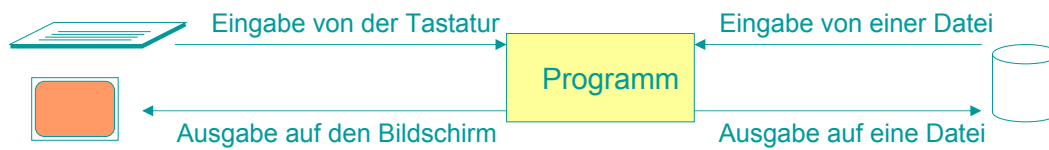
Arithmetischen Operationen lassen sich mit Zuweisung kombinieren

	<i>Kurzform</i>	<i>Langform</i>
$+=$	$x += y;$	$x = x + y;$
$-=$	$x -= y;$	$x = x - y;$
$*=$	$x *= y;$	$x = x * y;$
$/=$	$x /= y;$	$x = x / y;$
$\% =$	$x \% = y;$	$x = x \% y;$

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-linz.ac.at/Misc/JavaBuch>



Besonderheiten zur Eingabe

Eingabe von Tastatur

Eintippen von: 12 100  Return-Taste

füllt Lesebuffer

Programm:

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

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

Eingabe von Datei

kein Lesebuffer, *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();  
  
        ...  
    }  
}
```

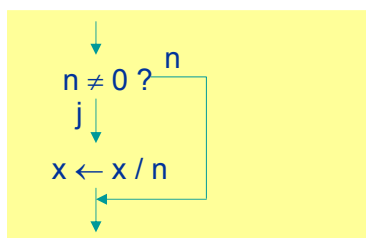
.... Übungsarbeit



Verzweigungen

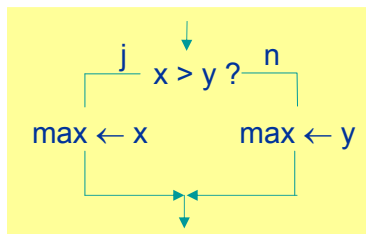


If-Anweisung



```
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 vorausgegangenem if.

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

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



Vergleichsoperatoren

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

	<i>Bedeutung</i>	<i>Beispiel</i>
==	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!
```



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)) ...
```



Kurzschlußauswertung

Zusammengesetzter Vergleich wird abgebrochen, sobald Ergebnis feststeht

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

wenn false, ist gesamter Ausdruck false

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

wenn true, ist gesamter Ausdruck true

äquivalent zu

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

```
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



Assertionen bei If-Anweisungen

```
if (condition)  
    // condition  
    ...  
else  
    // ! condition  
    ...
```

diese Assertion sollte man immer
hinschreiben oder zumindest im Kopf bilden

Beispiel: Maximum dreier Zahlen berechnen

```
int a, b, c, max;  
a = In.readInt();  
b = In.readInt();  
c = In.readInt();  
if (a > b)  
    if (a > c)  
        // a>b && a>c  
        max = a;  
    else  
        // a>b && c>=a  
        max = c;  
else  
    // b>=a  
    if (b > c)  
        // b>=a && b>c  
        max = b;  
    else  
        // b>=a && c>=b  
        max = c;  
Out.println(max);
```



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

```
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");
```

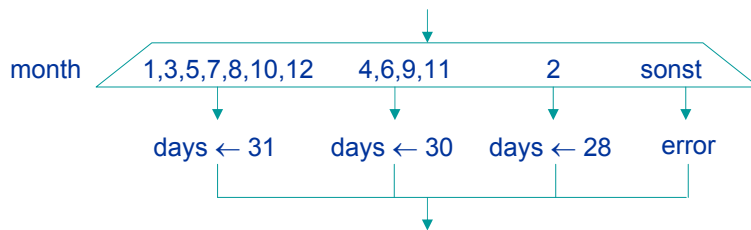


Beispielprogramm: If-Kaskade



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" ":".



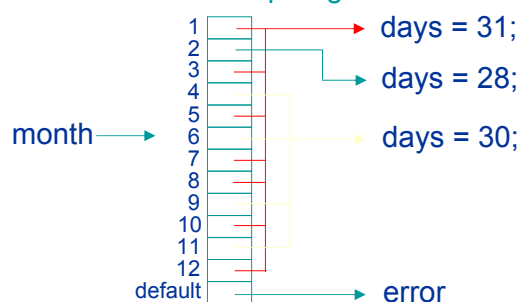
Unterschied zwischen If und Switch

```
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");
```

prüft Bedingungen sequentiell

```
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");
}
```

benutzt Sprungtabelle

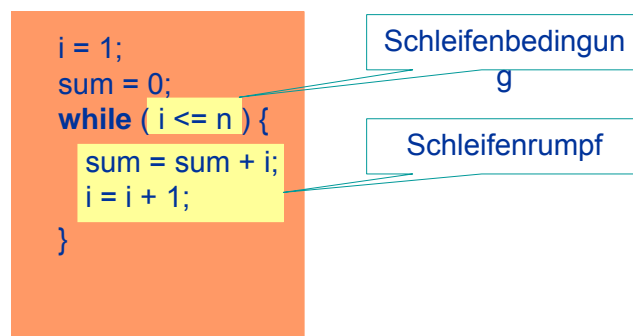
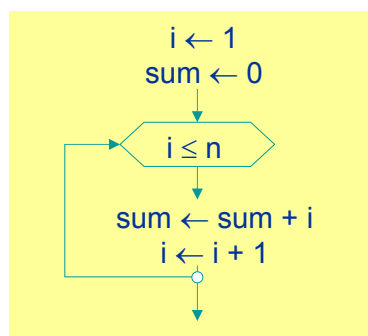


Schleifen



While-Schleife

Führt eine Anweisungsfolge aus, solange eine bestimmte Bedingung gilt



Syntax

Statement = Assignment | IfStatement | SwitchStatement | WhileStatement | ... | Block.

WhileStatement = **"while"** "(" Expression ")" Statement .

Wenn Schleifenrumpf aus mehreren Anweisungen besteht, muß er mit {...} geklammert werden.



Beispiel

Aufgabe: Zahlenfolge lesen und Histogramm ausgeben

Eingabe: 3 2 5

Ausgabe: ***
**

```
class Histogram {  
  
    public static void main (String[] arg) {  
        int i = In.readInt();  
        while (In.done()) {  
            int j = 1;  
            while (j <= i ) {Out.print("*"); j++;}  
            Out.println();  
            i = In.readInt();  
        }  
    }  
}
```

liest die Zahlenfolge

gibt i Sterne aus



Assertionen bei Schleifen

Triviale Assertionen

Aussagen, die sich aus der Schleifenbedingung ergeben

```
i = 1; sum = 0;  
while (i <= n) { /* i <= n */  
    sum = sum + i;  
    i = i + 1;  
}  
/* i > n */
```

sollte man immer hinschreiben oder
zumindest im Kopf bilden

Schleifeninvariante

Aussage über das berechnete Ergebnis, die in jedem Schleifendurchlauf gleich bleibt

```
i = 1; sum = 0;  
while (i <= n) { /* i <= n */  
    /* sum == Summe(1..i-1) */  
    sum = sum + i;  
    i = i + 1;  
}  
/* i > n */
```



Verifikation der Schleife

"Durchdrücken" der Invariante durch die Anweisungen des Schleifenrumpfs

```
i = 1; sum = 0;
while (i <= n) {
  /* sum == Summe(1..i-1) */
  sum = sum + i;
  sum' == sum + i
  sum == sum' - i == Summe(1..i-1)
  sum' == Summe(1..i)

  /* sum == Summe(1..i) */
  i = i + 1;
  i' == i + 1
  i == i' - 1
  sum == Summe(1..i'-1)
}
/* sum == Summe(1..i-1) */

/* i == n+1 && sum == Summe(1..i-1) ⇒ sum == Summe(1..n) */
```

Termination der Schleife muß auch noch bewiesen werden:

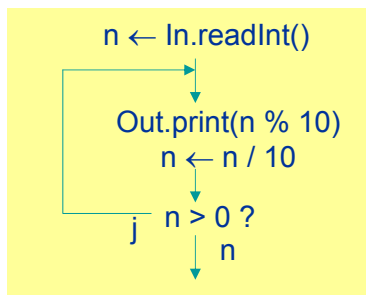
i wird in jedem Durchlauf erhöht und ist mit *n* beschränkt



Do-While-Schleife

Abbruchbedingung wird am Ende der Schleife geprüft

Beispiel: gibt die Ziffern einer Zahl in umgekehrter Reihenfolge aus



```
int n = In.readInt();
do {
  Out.print(n % 10);
  n = n / 10;
} while (n > 0);
```

Schreibtischtest

n	n % 10
123	3
12	2
1	1
0	

Syntax

Statement = Assignment | IfStatement | WhileStatement | DoWhileStatement | ... | Block.

DoWhileStatement = "do" Statement "while" "(" Expression ")" ";".

Wenn Schleifenrumpf aus mehreren Anweisungen besteht, muß er mit {...} geklammert werden.



Do-While-Schleife

Warum kann man dieses Beispiel nicht mit einer While-Schleife lösen?

```
int n = In.readInt();  
while (n > 0) {  
    Out.print(n % 10);  
    n = n / 10;  
}
```

"Abweisschleife"

Weil das für $n == 0$ die falsche Ausgabe liefern würde.
Die Schleife muß mindestens einmal durchlaufen werden, daher :

```
int n = In.readInt();  
do {  
    Out.print(n % 10);  
    n = n / 10;  
} while (n > 0);
```

"Durchlaufschleife"



For-Schleife

Falls die Anzahl der Schleifendurchläufe im voraus bekannt ist

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

- 1) Initialisierung der Laufvariablen
- 2) Abbruchbedingung
- 3) Ändern der Laufvariablen

Kurzform für

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



Syntax der For-Schleife

```
ForStatement = "for" "(" [ForInit] ";" [Expression] ";" [ForUpdate] ")" Statement.  
ForInit      = Assignment {"," Assignment}  
              | Type VarDecl {"," VarDecl}.  
ForUpdate    = Assignment {"," Assignment}.
```

Beispiele

```
for (i = 0; i < n; i++) ...
```

```
for (i = 10; i > 0; i--) ...
```

```
for (int i = 0; i <= n; i = i + 1) ...
```

```
for (int i = 0, j = 0; i < n && j < m; i = i + 1, j = j + 2) ...
```

```
for (;;) ...
```



Beispiel: Multiplikationstabelle drucken

```
class PrintMulTab {  
  
    public static void main (String[] arg) {  
        int n = In.readInt();  
        for (int i = 1; i <= n; i++) {  
            for (int j = 1; j <= n; j++) {  
                Out.print(i * j);  
                Out.println();  
            }  
        }  
    }  
}
```

Schreibischttest für n == 3

i	j	
1	1	1 2 3
	2	2 4 6
	3	3 6 9
2	4	
	1	
	2	
	3	
3	4	
	1	
	2	
	3	
4	4	
	1	
	2	
	3	



Abbruch von Schleifen

Beispiel: Summieren mit Fehlerabbruch

```
int sum = 0;
int x = In.readInt();
while (In.done()) {
    sum = sum + x;
    if (sum > 1000) {Out.println("zu gross"); break;}
    x = In.readInt();
}
```

break verläßt die Schleife, in der es enthalten ist (while, do, for)

Schleifenabbruch mit `break` möglichst vermeiden: schwer zu verifizieren. Meist läßt sich dasselbe mit `while` ebenfalls ausdrücken:

```
int sum = 0;
int x = In.readInt();
while (In.done() && sum <= 1000) {
    sum = sum + x;
    if (sum <= 1000) x = In.readInt();
}
// ! In.done() || sum > 1000
```



Abbruch äußerer Schleifen

Beispiel

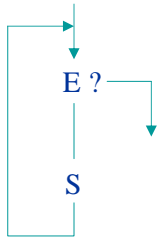
```
outer:                // Marke!
for (ii) {           // Endlosschleife!
    for (ii) {
        ...
        if (...) break; // verläßt innere Schleife
        else break outer; // verläßt äußere Schleife
        ...
    }
}
```

Wann ist ein Schleifenabbruch mit `break` vertretbar?

- bei Abbruch wegen Fehlern
- bei mehreren Aussprüngen an verschiedenen Stellen der Schleife
- bei echten Endlosschleifen (z.B. in Echtzeitsystemen)



Vergleich der Schleifenarten



Abweisschleife

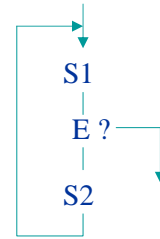
```
while (E)
  S
```

```
for (I; E; U)
  S
```



Durchlaufschleife

```
do
  S
while (E)
```



allgemeine Schleife

```
for (;;) {
  S1;
  if (E) break;
  S2;
}
```



Programm-Muster

Wie denken Programmierexperten?

- nicht in einzelnen Anweisungen
- sondern in größeren Programm-Mustern

Muster = Schema zur Lösung häufiger Aufgaben

Experten

- benutzen Muster intuitiv
 - z.B. Stellungen im Schachspiel
 - z.B. Redewendungen in Geschäftsbriefen
- lösen Aufgaben in Analogie zu bekannten Aufgaben

Frage: Was sind typische Muster in der Programmierung?



Zusammensetzen von Programmen aus Mustern

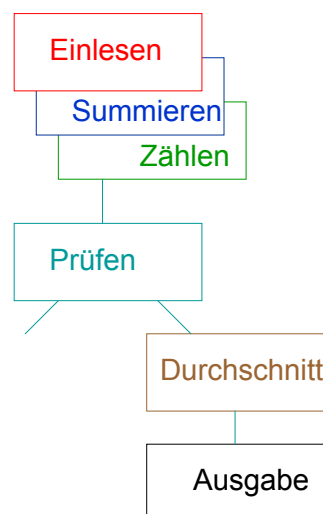
Beispiel: Berechnung des Durchschnitts einer Zahlenfolge

Einlese-Muster	Zähl-Muster
<pre>int x = In.readInt(); while (In.done()) { ... x = In.readInt(); }</pre>	<pre>int n = 0; while (...) { n++; ... }</pre>
Summierungs-Muster	Prüf-Muster
<pre>int sum = 0; while (...) { sum = sum + x; ... }</pre>	<pre>if (n != 0) ... else Out.println("error");</pre>
Durchschnitts-Berechn.	Ausgabe
<pre>float avg = (float)sum / n;</pre>	<pre>Out.println("avg = " + avg);</pre>



Zusammensetzen der Muster

```
int x = In.readInt();
int sum = 0, n = 0;
while (In.done()) {
    sum = sum + x;
    n++;
    x = In.readInt();
}
if (n != 0) {
    float avg = (float)sum / n;
    Out.println("avg = " + avg);
} else
    Out.println("error");
```



Weiteres Beispiel

Und-Verknüpfung von true- und false-Termen

true true true \Rightarrow true
true false true \Rightarrow false

Einlese-Muster	Und-Muster
<pre>boolean b = In.readBoolean(); while (In.done()) { ... b = In.readBoolean(); }</pre>	<pre>boolean ok = true; while (...) { ... ok = ok && b; }</pre>
Ausgabe-Muster	
<pre>Out.println(ok);</pre>	

```
boolean b = In.readBoolean();
boolean ok = true;
while (In.done()) {
    ok = ok && b;
    b = In.readBoolean();
}
Out.println(ok);
```



Schrittweise Eingabe von Ablaufstrukturen

Bei Schleifen

<pre>while (In.done()) { }</pre>
<pre>while (In.done()) { x = In.readInt(); }</pre>
<pre>while (In.done()) { ... x = In.readInt(); }</pre>

Bei Abfragen

<pre>if (n != 0) { } else { }</pre>
<pre>if (n != 0) { avg = (float)sum / n; Out.println("avg = " + avg); } else { }</pre>
<pre>if (n != 0) { avg = (float)sum / n; Out.println("avg = " + avg); } else { Out.println("error"); }</pre>



Entwurf, Implementierung und Testen von Programmen



Schritte des Algorithmenentwurfs und der Programmierung

- Problem erfassen und beschreiben (Aufgabenstellung)
- Lösungsidee erarbeiten und niederschreiben
- Lösungsidee in einen schrittweisen Ablauf überführen (= Algorithmus)
- Algorithmus so weit verfeinern und konkretisieren, dass Umsetzung in Java direkt möglich ist
- Programmierung in Java
- Überlegen von Testfällen (dabei alle möglichen Sonderfälle, Grenzfälle betrachten)
- Testen und Verbessern
- Dokumentieren der Ergebnisse



Beispiel "Die 2 größten Zahlen" (1)

▪ Aufgabenstellung

Entwickeln Sie einen Algorithmus, der folgendes leistet: Es sollen beliebig viele nicht-negative Zahlen von einem Eingabemedium gelesen werden. Die Eingabe ist beendet, wenn eine abschließende negative Zahl gelesen wird. Von den eingelesenen Zahlen sollen die zwei größten Werte berechnet und ausgegeben werden. Wenn keine oder nur eine Zahl eingegeben wurde, soll eine entsprechende Fehlermeldung ausgegeben werden.

Beispiele:

Eingabe: 1.5 2 0 4.25 1 -1
Ausgabe: Groesste Zahlen: 4.25 und 2
Eingabe: 5.25 -3.2
Ausgabe: Zuwenig Zahlen eingegeben!



Beispiel "Die 2 größten Zahlen" (2)

▪ Lösungsidee

Es werden nacheinander Werte eingelesen und die Anzahl der Zahlen mitgezählt. Ist eine Zahl negativ, so wird das Einlesen beendet. Ist die Anzahl der eingelesenen Werte (d.h. der nicht-negativen Werte) jedoch kleiner als 2, so wird eine Fehlermeldung ausgegeben, ansonsten werden die beiden bisher größten Werte ausgegeben.

Einlesen der Werte: Immer wenn ein gültiger Wert eingelesen wurde, wird er mit dem aktuellen größten und zweitgrößten Wert verglichen. Stellt man fest, dass man einen neuen größten oder zweitgrößten Wert, so werden größter und zweitgrößter Wert nachgeführt.

Da alle Zahlen größer gleich 0 sind, können die beiden größten Werte auf 0 initialisiert werden (minimale Maxima). Kommen später größere Zahlen vor, so sind sie auf jeden Fall größer gleich den Startwerten.

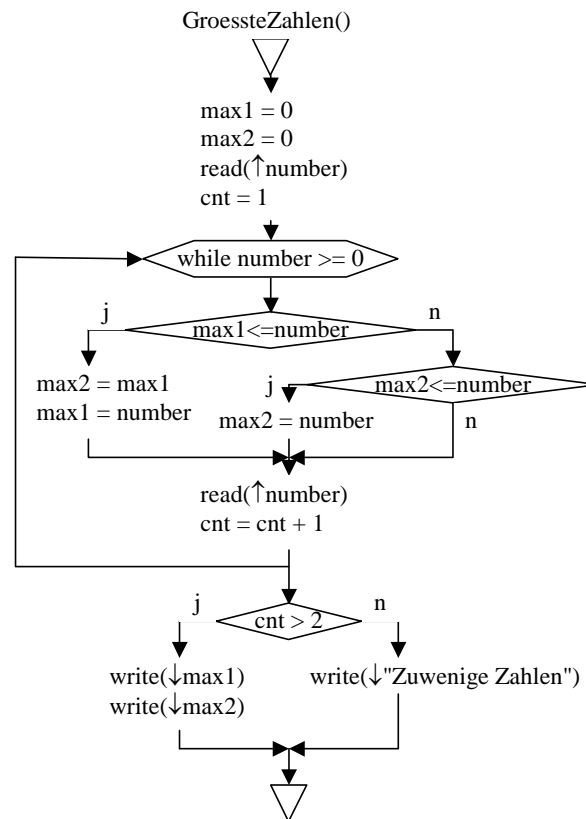
Beispiele:

größter: 20; zweitgrößter: 10; neuer: 15 => größter: 20; zweitgrößter: 15
größter: 20; zweitgrößter: 15; neuer: 10 => größter: 20; zweitgrößter: 15
größter: 20; zweitgrößter: 10; neuer: 30 => größter: 30; zweitgrößter: 20



Beispiel "Die 2 größten Zahlen" (3)

- Ablaufdiagramm



Beispiel "Die 2 größten Zahlen" (4)

- Programm: interaktiv entwickeln



- Was will man testen
 - Normale Fälle
 - testen normale Funktion des Algorithmus
 - dabei an alle unterschiedlichen Möglichkeiten denken
 - Grenzfälle
 - sind jene Fälle bei dem der Algorithmus noch funktionieren soll
 - Grenzfälle sind sehr wichtige Testfälle
 - Sonderfälle
 - sind jene Fälle, für die der Algorithmus eigentlich nicht funktioniert, die aber vorkommen können
 - benötigen spezielle Behandlung
- Wie will man testen
 - Eingabewerte
 - Testpositionen
 - erwartete Ergebnisse



Beispiel "Die 2 größten Zahlen" (5)

▪ Testfälle

Testfälle/Grenzen / Sonderfälle

Keine Zahl eingelesen => Fehlerfall, zuwenige Werte (minimale Eingabe)

Nur eine Zahl eingelesen => Fehlerfall, (gerade noch) zuwenige Werte

Zwei Zahlen eingelesen (minimaler Wert) => Grenzfall gerade genügend Werte

Eingabe einer Buchstabenfolge => Illegale Eingabe

Eingabe von ganzen und Fließkomma-Zahlen => Mögliche legale Eingaben

Eingabe einer Nullfolge => Minimale Maxima

Größter und zweitgrößter Wert gleich => Mehrfach vorkommendes Maximum

Größter und zweitgrößter Wert ungleich => Einfaches Maximum

Grosse Werte => Oberer Wertebereich

Viele Werte => Obere Werteanzahl

...



Beispiel "Die 2 größten Zahlen" (6)

- Testplan

Nr	Eingabewerte	Erwartete Ausgabe	Zweck
1	-1	Zuwenige Zahlen	Fehlerfall, zuwenige Werte
2	1 -2.5	Zuwenige Zahlen	Fehlerfall, zuwenige Werte (gerade noch)
3	5 7 -1	7 5	Grenzfall, gerade genügend Werte
4	Das ist ein Test	<undefiniert>	Illegale Eingabe
5	3 4.5 12.2 10 -2	12.2 10	Mögliche legale Eingaben
6	0 0 0 0 0 -1	0 0	Minimale Maxima
7	3 7.5 3.2 0 7.5 2 1.2 7 -1	7.5 7.5	Mehrfach vorkommendes Maximum
8	7 4 3 10 -1	10 7	Einfaches Maximum
9	1E100 1.53E123 13 -1	1.53E123 1E100	Oberer Wertebereich (?)
10	5 3 6 3 2 6 3 5 5.5 2 10 9 12 4 1.5 0.23 7.9 3.3 2.1 4 2 11 23 1.2 -1	23 12	Obere Werteanzahl (?)
11	0 2 3 4 23 123 -1	123 23	Aufsteigende Werte
12	123 23 4 3 2 0 -1	123 23	Absteigende Werte
13	4 23 2 2 123 2 0 3 -1	123 23	Allgemeiner Fall
14



Beispiel "Die 2 größten Zahlen" (7)

- Testen interaktiv entwickeln



Gleitkommazahlen



Die Typen float und double

Variablen

```
float x, y; // 32 Bit groß  
double z; // 64 Bit groß
```

Konstanten

```
3.14 // Typ double  
3.14f // Typ float  
3.14E0 // 3.14 * 100  
0.314E1 // 0.314 * 101  
31.4E-1 // 31.4 * 10-1  
.23  
1.E2 // 100
```

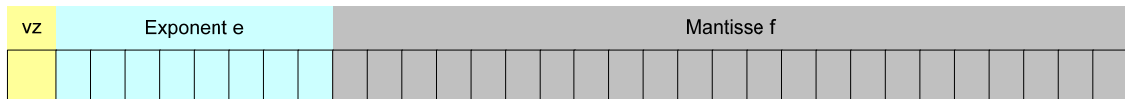
Syntax der Gleitkommakonstanten

FloatConstant	= [Digits] "." [Digits] [Exponent] [FloatSuffix].
Digits	= Digit {Digit}.
Exponent	= ("e" "E") ["+" "-"] Digits.
FloatSuffix	= "f" "F" "d" "D".



Gleitkommazahlen: Zahlendarstellung

- Darstellung von Gleitkommazahlen nach dem IEEE-Standard 754-1985
- Kodierung jeder float-Zahl binär in
 - 1 Bit Vorzeichen
 - 8 Bit Exponent
 - 23 Bit Mantisse



- Wert ergibt sich aus

$$(-1)^{vz} \times 2^{e-127} \times 1.f$$



Gleitkommazahlen: Zahlendarstellung

- Größte positive Zahl: $\approx 2 \times 2^{128} \approx 10^{38}$



- Kleinste positive Zahl: $\approx 2^{-23} \times 2^{-127} = 2^{-149} \approx 10^{-45}$



- Wert Null: $e = 0, f = 0$



- Unendlich: $e = 255, f = 0$



- NaN (Not a Number = undefiniert als Ergebnis von nicht gültigen Operationen)



Zuweisungen und Operationen

Zuweisungskompatibilität

`double` \supseteq `float` \supseteq `long` \supseteq `int` \supseteq `short` \supseteq `byte`

```
float f; int i;
f = i; // erlaubt
i = f; // verboten
i = (int)f; // erlaubt: schneidet Nachkommastellen ab; falls zu groß: maxint, minint
f = 1.0; // verboten, weil 1.0 vom Typ double ist
```

Erlaubte Operationen

- Arithmetische Operationen (+, -, *, /)
 - Vergleiche (==, !=, <, <=, >, >=)
- Achtung: Gleitkommazahlen sollte man nicht auf Gleichheit prüfen



Typen von Gleitkommaausdrücken

Der "kleinere" Operandentyp wird in der "größeren" konvertiert, zumindest aber in `int`.

`double` \supseteq `float` \supseteq `long` \supseteq `int` \supseteq `short` \supseteq `byte`

```
double d; float f; int i; short s;
...
d + i // double
f + i // float
s + s // int
```

Ein- / Ausgabe von Gleitkommazahlen

```
double d = In.readDouble();
float f = 3.14f; // es gibt in der Klasse In kein readFloat
Out.println("d = " + d + ", f = " + f);
```



Operatoren für Gleitkommazahlen

Addition:	+	: Gleitkomma × Gleitkomma	→ Gleitkomma
Subtraktion:	-	: Gleitkomma × Gleitkomma	→ Gleitkomma
Multiplikation:	*	: Gleitkomma × Gleitkomma	→ Gleitkomma
Division:	/	: Gleitkomma × Gleitkomma	→ Gleitkomma
Inkrement um 1.0:	++	: Gleitkomma	→ Gleitkomma
Dekrement um 1.0:	--	: Gleitkomma	→ Gleitkomma



Die Funktionsbibliothek Math

- Mit `Math` steht eine umfangreiche Funktionsbibliothek für mathematische Operationen zur Verfügung.
- Diese Operationen ruft man folgend auf

`Math.operator(argumente)`

sie liefern den entsprechenden Wert und können in Ausdrücken verwendet werden.

Beispiel: Sinusberechnung

```
double x;  
double sinusX;  
  
x = 30;  
sinusX = Math.sin(x);
```



Funktionen in Math (Auszug)

- `abs(double a)` Absolutbetrag von a
- `ceil(double a)` Kleinste Ganzzahl $\geq a$
- `floor(double a)` Größte Ganzzahl $\leq a$
- `max(double a, double b)` Maximum von a und b
- `min(double a, double b)` Minimum von a und b
- `log(double a)` natürlicher Logarithmus von a
- `pow(double a, double b)` a^b
- `round(double a)` Rundung auf `long`
- `cos(double a)` Kosinus
- `sin(double a)` Sinus
- `tan(double a)` Tangens
- `atan(double a)` Arkustangens
- ... und viele mehr

