Einfache Programme

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

```
Ziffer = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
Zahl =Ziffer {Ziffer}.
Gleitkommazahl = Zahl "." Zahl [ " E" [ "+" | " -" ] Zahl].
```

Grammatik der If- Anweisung

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

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 Programmteile 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
```

Zeichenketten

- 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  money
```

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)

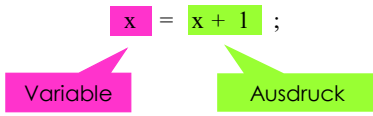
Jedenfalls: Deutsch und Englisch nicht mischen!!

Namenswahl für Variablen und Konstanten

Einige Tipps

- lesbar aber nicht zu lang
  z. B. *sum, phoneNumber*
- 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, streetNumber, expirationDate*
- mit Kleinbuchstaben beginnen,
  Worttrennung durch Großbuchstaben (camelCase)
  z. B. *inputText*
- Idealerweise Englisch

Zuweisungen (assignement)



1. Berechne den Ausdruck
2. Speichere einen 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

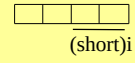
Beispiele

```
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
```

<- Achtung, schlechter Stil –- Increment und Decrement nie innerhalb eines Ausdrucks (expression) verwenden

Darf nur auf Variablen angewendet werden (nicht auf Ausdrücke).

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

Sollte nur als eingenständige 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

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 * 10 0
0.314E1              // 0.314 * 10 1
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"..
```

Zuweisungen und Operationen

Zuweisungskompatibilität

double ⊇ float ⊇ long ⊇ int ⊇ short ⊇ 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
f = 1.0f;    // ok
```

Erlaubte Operationen

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

Typen von Gleitkommaausdrücken

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

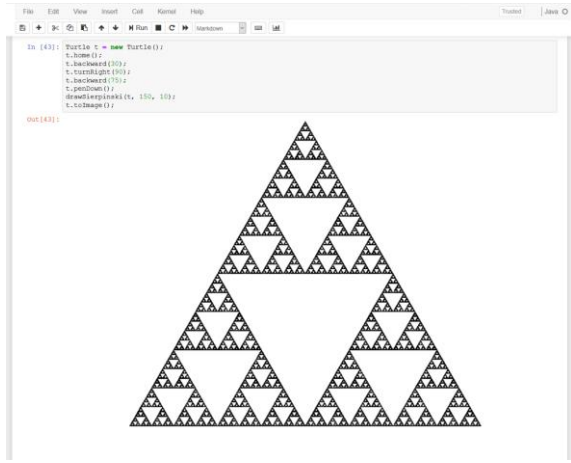
double ⊇ float ⊇ long ⊇ int ⊇ short ⊇ byte

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

Ein- / Ausgabe von Gleitkommazahlen

```
double d = Double.parseDouble(arg[0]);
float f = 3.14f;
System.out. println(" d = " + d + ", f = " + f);
```

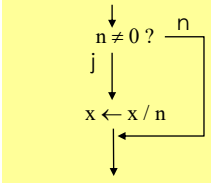
Interaktives Programmieren



Notebook: EinfacheProgramme.ipynb

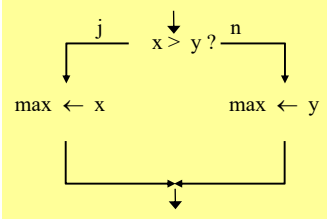
Verzweigungen

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 der then-Zweig oder der else-Zweig aus mehr als einer Anweisung besteht,
müssen sie durch { ... } geklammert werden.

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

Beispiel

```
if (x < 0) {
   negNumbers++;
   System.out.println(- x);
} else {
   posNumbers++;
   System.out.println(  x);
}
```

Einrückungen

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

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

```
if (x < 0) {
   negNumbers++;  System.out. println(- x);
} else {
   posNumbers++;   System.out.println(  x);
}
```

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

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*.

Empfehlung: Immer Klammern setzen

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

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)  System.out.println( " x is zero! " ) ;
```

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

```
if (x =0)  System.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

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

Datentyp wie **int**, nur eben mit den beiden Werten *true* und *false* .

Beispiel

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

Beachte

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

Negation zusammengesetzter Ausdrücke

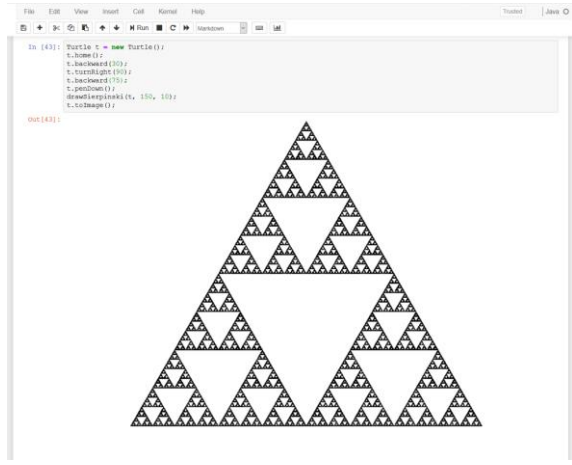
Regeln von DeMorgan

```
! (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
...
}
```

Interaktives Programmieren



Notebook: EinfacheProgramme.ipynb

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;
```

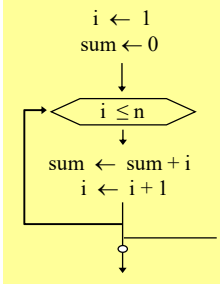
Was heißt "besser"?

- 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?

Schleifen

While- Schleife

Führt eine Anweisungsfolge aus, solange eine bestimmte Bedingung gilt.




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

Schleifenbedingung

Schleifenrumpf

Syntax

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

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

Ein Schleifenrumpf aus mehreren Anweisungen muß mit {...} geklammert werden.

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) */
```

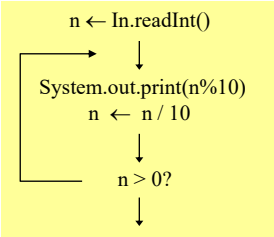
Termination der Schleife muß auch noch bewiesen werden:

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 {
   System.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 "**)**" ";".

Ein Schleifenrumpf aus mehreren Anweisungen muß mit {...} geklammert werden.

Do- While- Schleife

Warum kann man voriges Beispiel NICHT mit einer While - Schleife lösen?

```
int n = In. readInt();
while (n > 0) {
    System.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 {
    System.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 (;;) ...
```

Abbruch von Schleifen

Beispiel: Summieren mit Fehlerabbruch

```
int sum = 0;
int x = In. readInt();
while (In. done()) {
    sum = sum + x;
    if (sum > 1000) {System.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

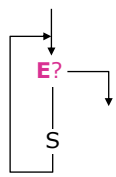
Beispiel

```
outer:                          // Marke!
for (;;) {                      // Endlosschleife!
    for (;;) {
        ...
        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



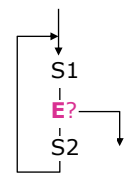
Abweisschleife

```
while(E)
   S

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

Durchlaufschleife

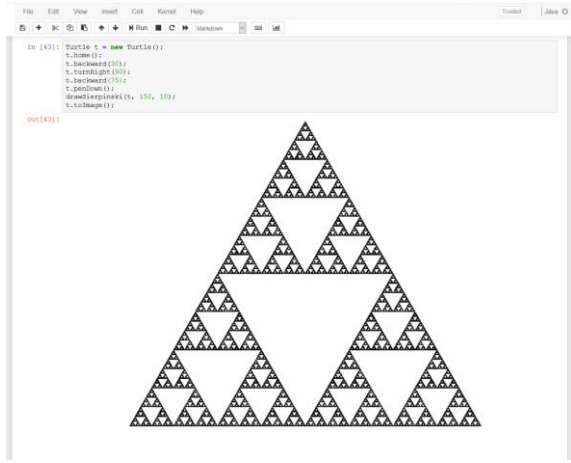
```
do
   S
while(E)
```



allgemeine Schleife

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

Interaktives Programmieren



Notebook: EinfacheProgramme.ipynb

SOFTWARE ENTWICKLUNG & KOMPILIEREN

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

int x = In. readInt();
while (In. done()) {
...
x = In. readInt();
}
	int n = 0;
	while (...) {
	n++;
	...
	}

Summierungs- Muster	**Prüf- Muster**

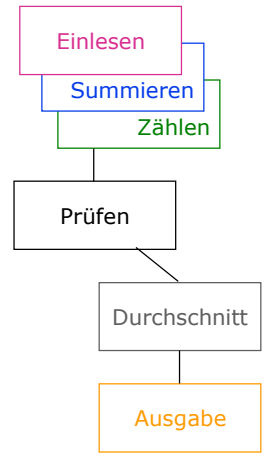
int sum = 0;
while (...) {
sum = sum + x;
...
}
	if (n != 0)
	...
	else
	System.out.println(" error");

Durchschnitts- Berechn.	**Ausgabe**
float avg = (float) sum / n;	**System.out.println(" avg = " + avg);**

Zusammensetzen von Programmen aus Mustern

Beispiel: Berechnung des Durchschnitts einer Zahlenfolge

```
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;
   System.out. println(" avg = " + avg);
} else
  System.out. println(" error");
```



Schrittweise Eingabe von Ablaufstrukturen

Bei Schleifen

```
while (In. done()) {
}
```

```
while (In. done()) {
x = In. readInt();
}
```

```
while (In. done()) {
...
x = In. readInt();
}
```

Bei Abfragen

```
if (n != 0) {
} else {
}
```

```
if (n != 0) {
   avg = (float) sum / n;
   System.out. println(" avg = " + avg);
} else {
}
```

```
if (n != 0) {
  avg = (float) sum / n;
  System.out. println(" avg = " + avg);
} else {
  System.out.println(" error");
}
```

Schrittweise Eingabe von Ablaufstrukturen

Bei Schleifen

```
while (In. done()) {
}
```

```
while (In. done()) {
x = In. readInt();
}
```

```
while (In. done()) {
...
x = In. readInt();
}
```

Bei Abfragen

```
if (n != 0) {
} else {
}
```

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

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