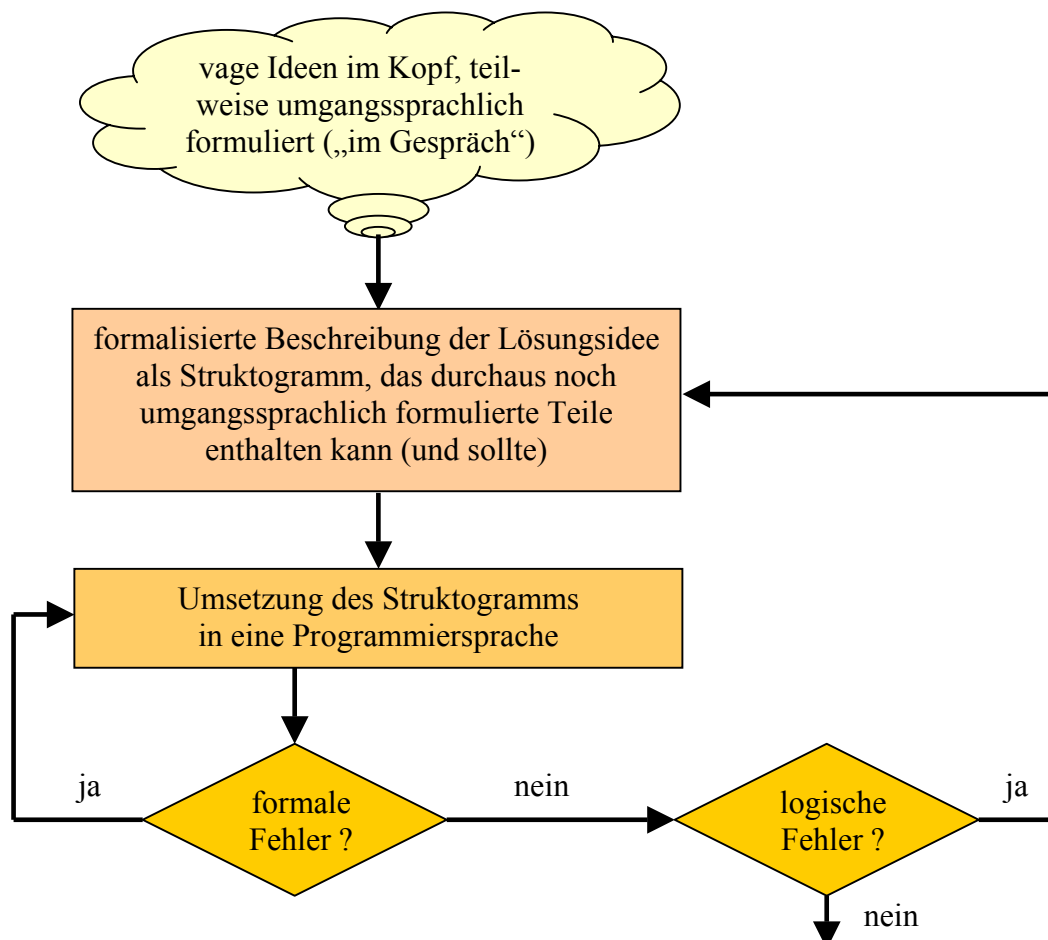


# Java-Anweisungen und -Ausdrücke, Syntaxdiagramme und Struktogramme

## 1. Bezug zum Unterricht: Leistungsmessung

Der Informatikunterricht *besteht nicht* aus einem Programmierkurs in einer bestimmten Programmiersprache – einerseits. Andererseits bezieht der Informatikunterricht seinen Reiz für alle Beteiligten in hohem Maße aus der Möglichkeit, dass Schülerinnen und Schüler selbständig Probleme angehen und (hoffentlich wenigstens teilweise) lösen, *eigene Ideen* formulieren und vor allem *realisieren* können, und dafür müssen sie recht gut programmieren können. Der Informatikunterricht *enthält* deshalb einen Programmierkurs *im notwendigen Umfang*, und der ist nicht unerheblich, denn sonst bleibt die Selbständigkeit der Schülerinnen und Schüler bloße Theorie. Der notwendige Umfang ergibt sich natürlich aus EPAs, Rahmenrichtlinien usw. (mit allen ihren Freiheiten), vor allem aber aus den bearbeiteten Problemstellungen. Ziel ist es nicht, eine bestimmte Programmiersprache möglichst vollständig zu erlernen, sondern Problemlösungsmethoden kennen zu lernen und programmiersprachenfrei zu formulieren, die dann in einer geeigneten Sprache realisiert werden.

Da die Unterrichteten das Problemlösen erst erlernen, ist es für die *Leistungsmessung* nicht vertretbar, vollständige Problemlösungen (als eine Art K-O-System) zu erwarten. Notwendig sind Aufgabenstellungen, in denen Teile dieses Wegs gegangen werden. Verdeutlichen wir uns den Problemlösungsprozess, dann kommen wir wahrscheinlich zu einem Ergebnis, das in etwa so aussieht:



Teile dieses Prozesses können durch folgende Aufgabenarten nachgebildet werden:

**1. Formulierung einer Lösungsidee als Struktogramm**

Anmerkung: *Meist schwierig, weil problemlösendes Denken erforderlich ist. Kann vom Umfang her nur einen kleinen Teil der Leistungsmessung ausmachen.*

**2. „Trockentest“ (trace) eines gegebenen Struktogramms**

Anmerkung: *Einfach. Das Ergebnis kann oft eine Tabelle sein, die die zeitlich veränderte Belegung der Variablen darstellt.*

**3. Beweis der Korrektheit eines gegebenen Struktogramms**

Anmerkung: *Meist schwierig, weil ein eigener Lösungsansatz gefunden werden muss und Kenntnisse in mathematischer Beweisführung („vollständige Induktion“) erforderlich sind.*

**4. Übersetzung eines gegebenen Struktogramms in eine Programmiersprache**

Anmerkung: *Mittlerer Schwierigkeitsgrad, weil formale Zusätze („Variablen- und Typvereinbarungen, ...“) gemacht werden müssen. Gut kombinierbar mit (1.), wenn das Struktogramm nicht alle Details vollständig beschreibt. Damit sind alle Anforderungsbereiche in einer Aufgabe zu vereinen.*

**5. Syntaxfehlersuche in einem gegebenen Programmlisting**

Anmerkung: *Eher einfach.*

**6. Suche logischer Fehler in einem Programmlisting oder Struktogramm**

Anmerkung: *Mittlerer bis hoher Schwierigkeitsgrad, je nach Vorerfahrungen und Aufgabenstellung. Kann gut mit (2.) kombiniert werden, so dass alle Anforderungsbereiche in einer Aufgabe zu finden sind.*

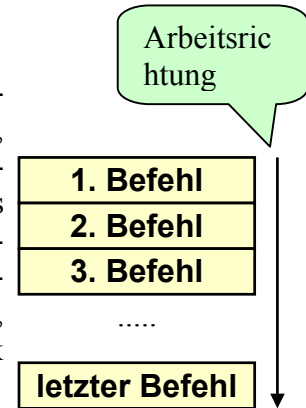
Für alle diese Aufgaben sind Kenntnisse der grundlegenden Struktogrammbestandteile erforderlich. Syntaxdiagramme legen die Syntax einer Programmiersprache eindeutig fest und sind wichtige Hilfsmittel im theoretischen Teil der Kursfolge. Sie sollten dazu früh eingeführt werden. Programmlistings sind Instanzierungen von Struktogrammen, die mit Hilfe der Syntaxdiagramme überprüft werden können.

Die Kästchenschreibweise der Struktogramme dient dazu, Vorüberlegungen zu Programmen übersichtlich aufzuschreiben, ohne sich um die genauen Einzelheiten der benutzten Programmiersprache zu kümmern (obwohl man sie natürlich im Hinterkopf hat). In den Kästchen stehen also mehr oder weniger ausführliche Anmerkungen zu dem, was an dieser Stelle geschehen soll. Weiß man schon, wie das Gewünschte realisiert werden kann, dann beschränkt man sich auf wenige Schlagworte. Will man also z. B. einen bewegten Ball darstellen, dann schreibt man:

<b>Zeichne einen Ball</b>
<b>Lösche ihn wieder</b>
<b>verändere die Ballposition</b>
usw.

## 2. Struktogramme und Java-Syntax

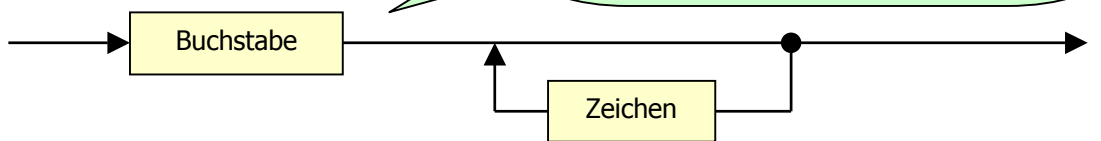
Meist beschreiben wir mit **Struktogrammen** nicht vollständige Programme, sondern **Programmteile**, also Funktionen, Prozeduren oder Ereignisbehandlungsmethoden. Solche Programmstücke werden von Befehlsfolgen gebildet, die wir uns – symbolisiert durch Kästchen – aufeinandergestapelt vorstellen können. Der Kästchenstapel wird von oben nach unten durchlaufen. Ein neues Kästchen wird erst dann betreten, wenn das vorherige verlassen wurde. Das Programmstück endet mit dem letzten Kästchen



**Syntaxdiagramme** bestehen aus Graphen, also Kanten (Pfeile) und Knoten (Kästchen bzw. Ellipsoide), die Teile der Programmiersprachensyntax beschreiben. Ein syntaktisch richtiges Programmstück stellt einen möglichen Weg durch den Graphen dar. Fehlerhafte Programme führen in Sackgassen. Trifft man auf dem Weg auf ein **Ellipsoid**, dann stellt dieses einen Schlüsselbegriff der Programmiersprache dar, der nicht weiter ersetzt werden kann. **Kästchen** enthalten Sprachelemente, die in anderen Syntaxdiagrammen noch weiter zu präzisieren sind.

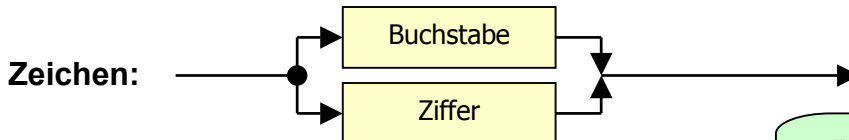
**Beispiel:** Java-Bezeichner (Namen)

**Bezeichner:**

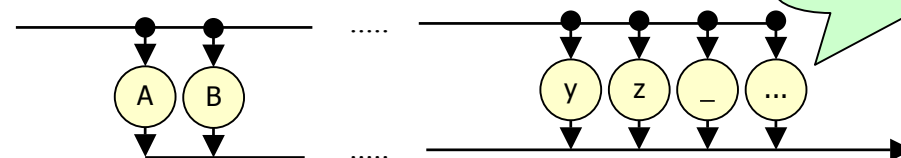


zu lesen:  
 Ein Bezeichner beginnt mit einem Buchstaben. Ggf. werden weitere Zeichen angehängt, wobei noch offen ist, was unter Buchstaben und Zeichen zu verstehen ist.

**Zeichen:**

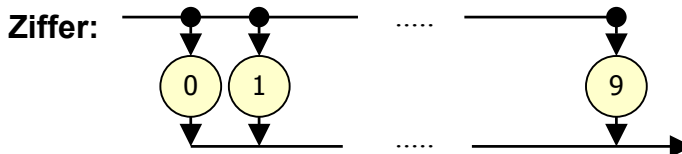


**Buchstabe:**



... und andere Unicode-Zeichen wie z. B. Währungssymbole

**Ziffer:**



**Beispiele:**

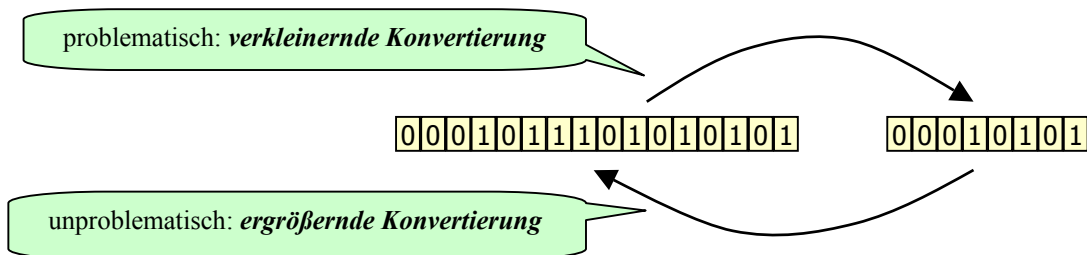
- für korrekte Bezeichner: a, A1, \_A1, Betrag\_in\_\$, MeinErstesProgramm, ...
- für falsche Bezeichner: 1, 1A, Klammer@ffe, Mein Erstes Programm, ...

Zahlreiche Bezeichner sind für Java selbst reserviert. Die darf bzw. sollte man nicht für andere Zwecke missbrauchen. Man vermeidet Namenskonflikte weitgehend, wenn man deutsche Bezeichnungen wählt (weil Java englische benutzt).

## 2.1 Einfache Datentypen und Typumwandlungen

Java ist eine Sprache mit strenger Typprüfung, d. h. bevor eine Variable einen Wert erhält, wird geprüft, ob dieser Wert auch passt. Jeder Datentyp beansprucht einen gewissen Platz im Speicher. Speichern wir die ganze Zahl „3“ in einer **short**-Variablen, dann beansprucht sie 8 Bit, in einer **long**-Variablen aber 64 Bit, obwohl sich ihr Wert nicht ändert. (Der überflüssige Platz wird mit Nullen aufgefüllt.) Umgekehrt passt ein **long**-Wert, wenn er seinen Platz ausschöpft, nicht mehr in eine **short**-Variable. Da die konkreten Werte, die Variable während des Programmlaufs annehmen, vor dem Programmlauf (meist) nicht bekannt sind, ist es sinnvoll, schon bei der Übersetzung solche möglichen Überlauffehler zu erkennen und damit zu verhindern.

Wenn ein kleiner Typ, z. B. **short**, in einen großen, z. B. **long**, umgewandelt werden muss (z. B. bei der Addition eines **short**- und eines **long**-Wertes), dann kann das automatisch passieren, weil der Inhalt des kleinen in jedem Fall Platz im großen hat. (Es handelt sich um eine **vergrößernde Konvertierung**.) Da die Lage völlig klar ist, wird von Java eine **automatische Typumwandlung** veranlasst. Im umgekehrten Fall – bei einer verkleinernden Konvertierung – werden Bits freigegeben und deren Inhalte gehen verloren. Dadurch können Daten verändert werden, so dass diese Konvertierung im Normalfall verboten ist.



Sollen trotz Typproblemen Werte eines Typs Variablen eines anderen zugewiesen (oder in Ausdrücken benutzt) werden, dann kann man eine Typumwandlung erzwingen, indem man den gewünschten Datentyp vor einem Wert oder einem Variablenbezeichner – in runde Klammern gesetzt – angibt (**type-casting**). Ein Standardfall in Java dafür ist die Benutzung von Zufallszahlen, die Fließkommazahlen vom Typ **double** produziert. Rundet man diese, dann erhält man ganze Zahlen vom Typ **long**. Zum Zeichnen auf dem Bildschirm benötigt man aber ganze Zahlen vom Typ **int**. Will man also Zufallszahlen z. B. für Zufallsgrafiken benutzen, dann muss man einen **Cast** durchführen, um **long**-Zahlen in **int**-Zahlen umzuwandeln.

```
int i = (int) Math.round(Math.random()*100);
```

An einfachen Datentypen stehen zur Verfügung: (Einfache Typen sind **atomar**, also nicht weiter strukturiert – wenn man davon absieht, dass sie alle durch Bitfolgen repräsentiert werden.)

Typ	Inhalt	Größe	Wertebereich
boolean	Wahrheitswert	1 Bit	false, true
char	Unicode-Zeichen	16 Bit	„fast alles, was man schreiben kann“
byte	ganze Zahl mit Vorzeichen	8 Bit	-128 bis 127
short	ganze Zahl mit Vorzeichen	16 Bit	-32768 bis 32767
int	ganze Zahl mit Vorzeichen	32 Bit	-2147483648 bis 2147483647
long	ganze Zahl mit Vorzeichen	64 Bit	ausreichend groß
float	Fließkommazahl IEEE754	32 Bit	±1.4E-45 bis ±3.4028235E+38
double	Fließkommazahl IEEE754	64 Bit	±4.9E-324 bis ±1.797693E+308

## 2.2 Literale, Variable, Ausdrücke und Operatoren

Schreibt man einen *Wert* in der üblichen Form hin, dann erhält man ein *Literal*. (Beispiele: -1.23456 oder true) Bezeichnet man einen Speicherbereich mit einem symbolischen Namen, der den Konventionen für *Bezeichner* gehorchen muss, dann haben wir eine *Variable*. Die für eine Variable erforderliche Speichergröße ergibt sich aus ihrem *Typ*. Wir können einer Variablen einen Wert zuordnen, indem wir ihr z. B. ein Literal zuweisen, das ihrem Typ entspricht (oder automatisch in diesen umgewandelt werden kann). (Beispiele: `int i = 23;` oder `double r=22.3;`)

Etwas kompliziertere Konstrukte können mit Hilfe von Ausdrücken formuliert werden. Ein *Ausdruck* wird vom Java-Interpreter so ausgewertet, dass er einen Wert ergibt. Dazu werden Variable durch ihre momentanen Werte ersetzt und zusammen mit den unveränderten Literalen unter Beachtung der *Operatoren* und ihrer *Priorität* ausgewertet. Gelten keine anderen Regeln, dann werden Ausdrücke von links nach rechts fortschreitend berechnet.

### Beispiele:

- `1+2+3;`
- `(a+4)/(a+2);`
- `Math.sqrt(1-x*x)+1;`
- `(a==1 && b>=0) || (c !=4)`

In Java gibt es sehr viele Operatoren für unterschiedliche Datentypen, deren Wirkungsweise und deren Priorität manchmal nicht sofort zu verstehen ist. Man sollte deshalb immer – eventuell auch überflüssige – Klammern setzen, um das gewünschte Ergebnis eindeutig zu definieren. Ich verweise hierzu auf die einschlägige Literatur und gebe nur eine kurze Übersicht über „normale“ Operatoren an:

### Arithmetische Operatoren:

Operator	Operation	Beispiel(e)
-	unäres Minus	-3    -b
+	Addition	3+4    b+2.345
-	Subtraktion	2-3    a-b-3.1415
*	Multiplikation	2*3    3*a*b
/	Division	2/3    a/2
%	Modulo (Rest)	7%2    a%b

### Vergleichsoperatoren:

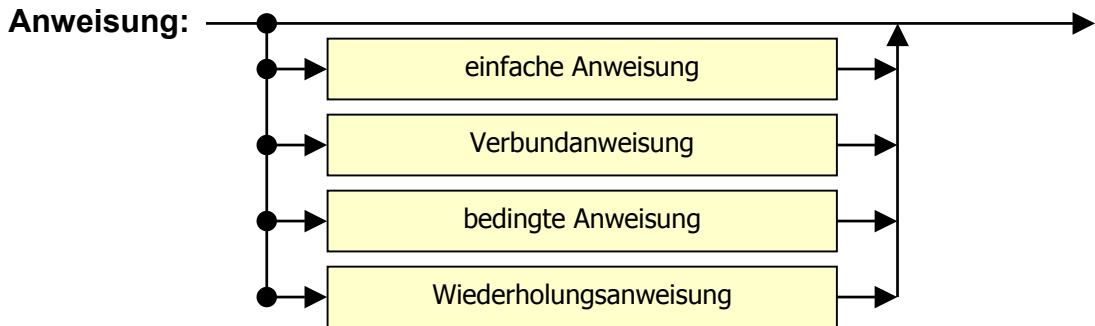
Operator	Operation	Beispiel(e)
==	gleich	if (a == 3) ...
!=	ungleich	if (x != 0) ...
<	kleiner	if (a < b) ...
<=	kleiner gleich	if (a <=100) ...
>	größer	if (a > b) ...
>=	größer gleich	if (a >= b) ...

### boolesche Operatoren:

Operator	Operation	Beispiel(e)
&&	logisches UND	if (x<10 && x>0) ...
&	bitweises UND	i & 2 (funktioniert auch als logisches UND)
	logisches ODER	if (x==0    x <0) ...
	bitweises ODER	i   2 (funktioniert auch als logisches ODER)
!	NICHT	if (!(x>1)) ...
^	Exklusiv-ODER	a ^ b (funktioniert auch bitweise als XOR)

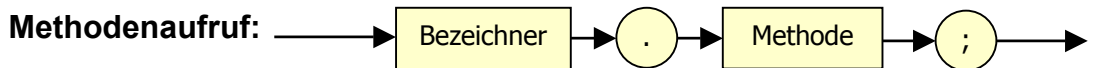
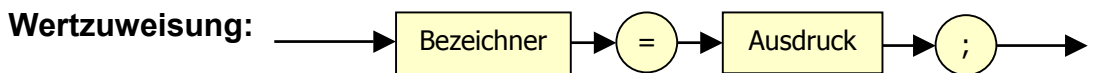
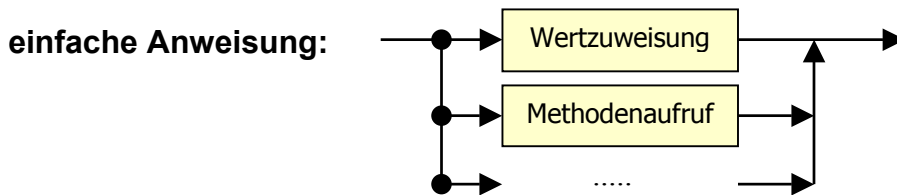
### 2.3 Anweisungen

Man unterscheidet unterschiedliche Anweisungsarten, wobei die meisten nicht Aktionen auslösen, sondern den Programmfluss steuern, also festlegen, in welcher Reihenfolge die Befehle auszuführen sind. In den verschiedenen Programmiersprachen werden diese Befehle syntaktisch unterschiedlich formuliert, ähneln sich aber stark.



**Einfache Anweisungen:** einfache Anweisung

Einfache Anweisungen werden durch einen einzelnen Kasten symbolisiert. Sie lassen sich in einem Stück ausführen. Beispiele sind Wertzuweisungen und Methodenaufrufe.



Aus einem Ausdruck muss zur Zeit der Ausführung ein eindeutiger Wert ermittelt werden können, der z. B. dem Typ der Variablen entspricht, die durch den Bezeichner benannt ist.

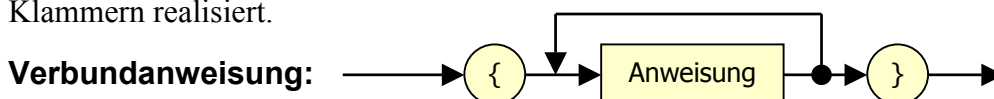
**Beispiele:**

- für Wertzuweisungen: `a = 5;`                      `auto.farbe = new Color(255,90,50);`
- für Methodenaufrufe: `auto.ZeigeDich();` `g.drawRect(x,y,100,75);`

**Verbundanweisungen:**



Befehlsfolgen lassen sich zu einem einzigen Befehl zusammenfassen, wenn man sie zusammenklammert. Sie stellen dann eine einzige Verbundanweisung dar. Verbundanweisungen treten meist in Zählschleifen oder bedingten Anweisungen auf, bilden aber auch den Rumpf einer Methode. In Java wird die Klammerung durch geschweifte Klammern realisiert.



**Beispiele**

- für eine Verbundanweisung: `{ a = 5; b=10; g.drawLine(0,0,a,b); }`
- für Verbundanweisungen in einer Alternative:
 

```
if (x !=0) { y1=1/x; y2=-1/(x*x); }
else { g.drawString("... so geht das nicht!",2,2); break; }
```

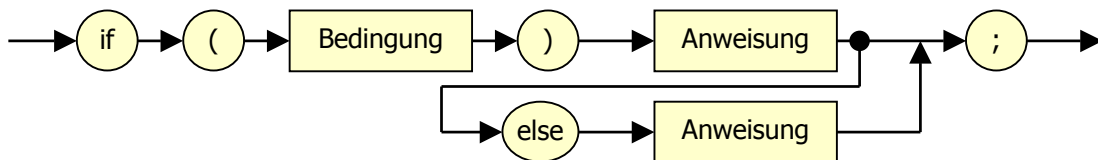
**Bedingte Anweisungen:**

In bedingten Anweisungen wird eine Befehlsfolge nur dann ausgeführt, wenn eine Bedingung erfüllt ist. Gibt es nur zwei Möglichkeiten, dann wählt man die **Alternative**, gibt es mehrere, dann die **Mehrfachauswahl**.

Bei der Alternative unterscheidet man die einseitige und die zweiseitige Alternative. Im ersten Fall geschieht nur dann etwas, wenn die Bedingung erfüllt ist.

Diese Aussage ist	
wahr	falsch
hier steht, was geschieht, wenn die <b>Bedingung erfüllt</b> ist	hier steht, was geschieht, wenn die <b>Bedingung nicht erfüllt</b> ist (ggf. nichts)

**Alternative:**



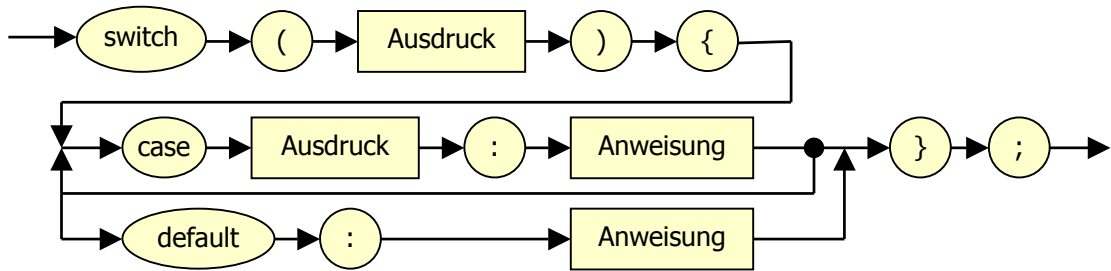
**Beispiele für Alternativen:**

- `if (x==0) g.drawString("x ist Null!",2,2);`
- `if (name=="Peter") g.drawString("Hallo Peter!",2,2);`  
`else g.drawString("Hallo",2,2);`
- `if (x==0||x>100) g.drawString("falscher Wert!",2,2);`

In Bedingungen können die **booleschen Operatoren** benutzt werden.

Die Mehrfachauswahl wird eigentlich nicht unbedingt benötigt, weil sie sich auch durch eine Verschachtelung von Alternativen realisieren lässt. Sie erleichtert allerdings oft die Schreibweise und macht die Programmierung damit übersichtlicher und eleganter.

Falls dieser Ausdruck den folgenden Wert hat				
Wert 1:	Wert 2:	Wert 3:	...	sonst:
tue dies	tue das	tue jenes	... :	tue sonst-was
			...	

**Mehrfachauswahl:****Beispiel** für eine Mehrfachauswahl:

- `switch(i)`

```

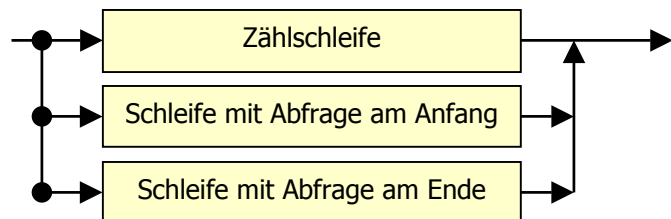
{
  case 1: return "Das Ergebnis ist 1"; break;
  case 2: return "Das Ergebnis ist 2"; break;
  case 3: return "Das Ergebnis ist 3"; break;
  default: return "Das Ergebnis ist größer als 3";
}

```

Man beachte die **break**-Anweisungen im obigen Beispiel. In der Mehrfachauswahl werden die unterschiedlichen **case**-Ausdrücke zwar direkt angesprochen, **aber nicht wieder verlassen**. Ohne **break**-Anweisungen würden alle auf den Einsprungpunkt folgenden Anweisungen in den restlichen **case**-Klauseln ebenfalls ausgeführt. Man muss also selbst dafür sorgen, dass nach Ausführung der gewünschten Befehlsfolge die **switch**-Anweisung verlassen wird.

**Wiederholungsanweisungen:**

Wiederholungsanweisungen oder **Schleifen** werden als einzelne Anweisungen aufgefasst, die (meist) mehrere Befehle enthalten, die ggf. mehrfach wiederholt werden. Man unterscheidet meist drei Schleifenarten:

**Wiederholungsanweisung:**

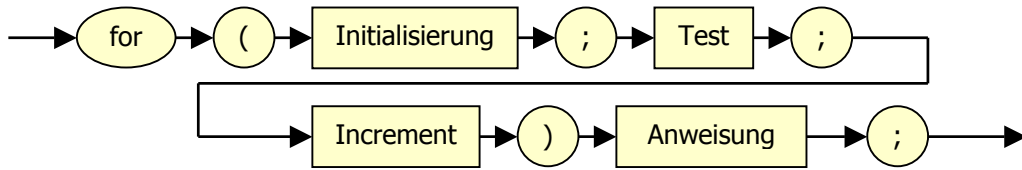
In einer **Zählschleife** ist die Zahl der Wiederholungen vor Eintritt in die Schleife bekannt und wird meist von einer **Zählvariablen** kontrolliert. Die Zählvariable ist oft vom Typ `int` und wird in der Schleife schrittweise verändert. (Tatsächlich können Zählschleifen bedeutend komplizierter aufgebaut sein!)

**FÜR** zählvariable **VON** anfangswert **BIS** endwert **TUE**

alles, was hier steht

Typische Einsatzgebiete für Zählschleifen sind das Durchlaufen eines Feldes oder einer Zeichenkette, wobei immer die gleichen Operationen mit allen Elementen dieser Strukturen ausgeführt werden, sowie Sortier- und Suchvorgänge.

**Zählschleife:**



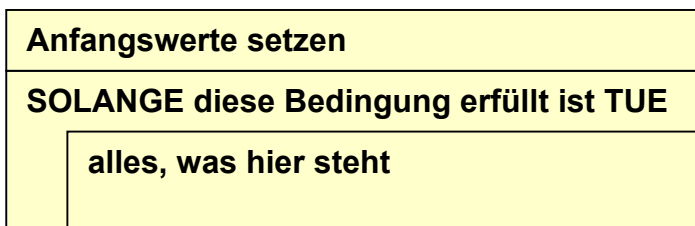
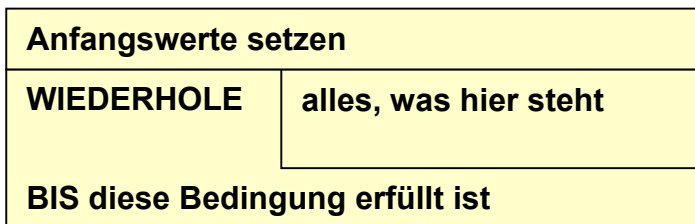
Die **Initialisierung** besteht meist darin, der Zählvariablen einen Anfangswert zuzuweisen. Bei Bedarf kann sie an dieser Stelle auch direkt vereinbart werden. Im **Test** wird vor jedem Schleifendurchlauf überprüft, ob die Schleife fortgesetzt werden soll. Im **Inkrement** wird der Wert der Zählvariablen verändert, meist mit einem der Operatoren ++ oder --, die den Wert der Zählvariablen schrittweise erhöhen oder verringern. Die Anweisung i++ ist äquivalent zu i=i+1.

**Beispiele** für Zählschleifen:

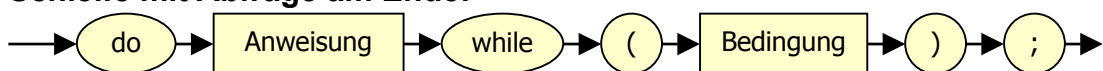
- for(int i=1; i<10;i++) summe=summe+i;
- for(j=100;j>0;j--) vertausche(feld[j],feld[j+1]);
- for(int i=1;i<=nmax||gefunden;i++) if(feld[i]>maximum) gefunden=true;

Wie gesagt: Man kann mit Zählschleifen auch wesentlich komplexere Konstrukte erzeugen. Da diese aber oft schwer zu durchschauen und somit fehlerträchtig sind und außerdem die anderen Schleifenarten hier übersichtlichere Formulierungsmöglichkeiten bieten, sollte man den Einsatz von Zählschleifen auf deren typische Anwendungen beschränken.

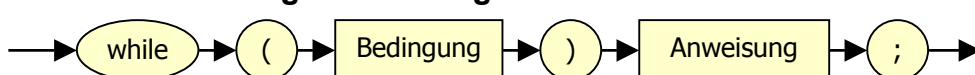
Kennt man die Zahl der Wiederholungen vor Eintritt in die Schleife nicht oder kann der Abbruch der Schleife durch unterschiedliche Ereignisse verursacht werden, dann benutzt man entweder eine **Schleife mit Abfrage am Ende** oder eine **Schleife mit Abfrage am Anfang**. Meist müssen vor Eintritt in die Schleife bestimmte **Anfangswerte** gesetzt werden. Am Ende bzw. Anfang der Schleife befindet sich die **Abbruchbedingung**, die bei jedem Schleifendurchlauf überprüft wird.



**Schleife mit Abfrage am Ende:**



**Schleife mit Abfrage am Anfang:**



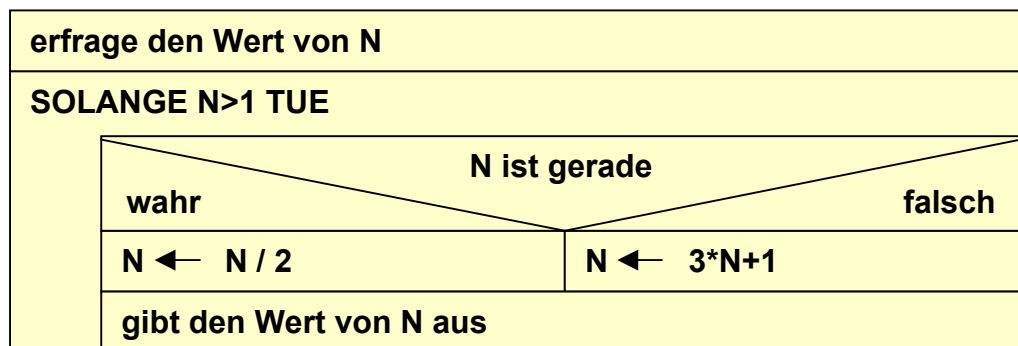
**Beispiele** für diese Schleifen:

- `i=1; while(i<10) i++;`
- `summe=0; i=1; do {summe=summe+i; i=i+1;} while(i<100);`
- `summe=0; i=1; while(i<100) {summe=summe+i; i=i+1};`

Der Programmfluss in Schleifen kann zusätzlich durch `break`- und `continue`-Befehle beeinflusst werden. Nach einem `break` wird der aktuelle Block verlassen (s. `switch`), nach einem `continue`-Befehl wird direkt der nächste Schleifendurchlauf begonnen, ohne die restlichen Anweisungen des Blocks auszuführen. Beide Befehle eignen sich dazu, kompliziertere Schleifenarten aufzubauen. Da hierdurch die Programmstruktur unübersichtlicher wird, sollte weitgehend auf diese Möglichkeiten verzichtet werden.

### 3. Aufgaben

1. Gegeben ist das folgende Struktogramm:



- a: Machen Sie einen „Trockentest“ (trace) des Programms in Form einer Tabelle, die nacheinander die von der Variablen N angenommenen Werte enthält.
  - b: Übersetze Sie das Struktogramm in Java.
  - c: Testen Sie das Programm für unterschiedliche Anfangswerte von N. Terminiert es immer?
2. Gegeben ist das folgende Stück Java-Code:

```
integer i, j, summe
i:=1; sum=0;
for (int i=1, j<100, i++)
    j=i+1;
    summe=summe+i;
```

- a: Suchen Sie mit Hilfe der Syntaxdiagramme von Java alle **formalen Fehler**. Versuchen Sie mit Hilfe der Punkte, an denen Fehler „sichtbar“ werden, geeignete Fehlermeldungen zu produzieren. Vergleichen Sie Ihre Ergebnisse mit denen des Java-Compilers, indem Sie den Code von der Maschine überprüfen lassen.
- b: Das Programmstück sollte eigentlich u.a. die Summe der ersten 100 Zahlen berechnen. Versuchen Sie alle **logischen Fehler** des Programms (nach Korrektur der formalen) zu finden. Überprüfen Sie auch jetzt Ihre Ergebnisse mit dem Computer.