Erinnerung:

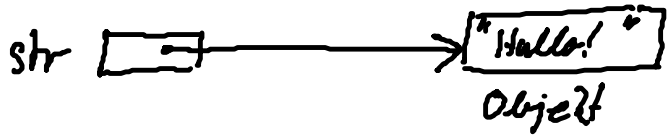
8 Standardtypen → über Variablenname wird der Wert angesprochen.

alles andere:

Referenztypen → über Variablenname wird Speicheradresse (Referenz) für das Objekt angesprochen.

```
String str = new String("Hallo!");
```

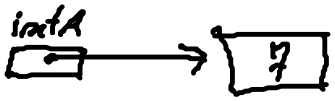
String: Klasse/Typ; str: Variable vom Typ String; String("Hallo!"): Konstruktor der Klasse String



bei Referenztypen benötigen wir Methoden, um mit Objekten zu arbeiten. z.B. str.length();

Wrapperklassen für Standardtypen (z.B. Integer für int) stellen Methoden zur Verfügung.

```
Integer intA = new Integer(7);
Integer intA = 7;
```



Anweisungen gleichbedeutend. Speicherbild gleich.

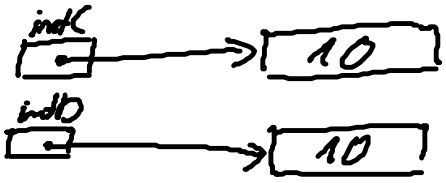
↖ boxing (Standardtyp → Wrapperklasse)

Implizite Typumwandlung (Casting) zwischen Standardtyp und der zugehörigen Wrapperklasse. "Auto boxing"

```
Integer intB = 8;
```

```
int c = intA + intB;
```
↑ unboxing (Wrapperklasse → Standardtyp)

```
Integer intC = new Integer(10);
Integer intD = new Integer(10);
```



`intC == intD` ergibt false
`intC.equals(intD)` ergibt true

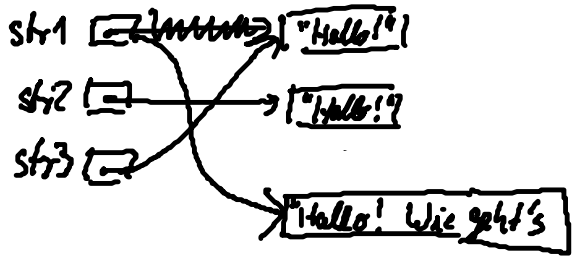
⚠ Empfehlung: Bei Wrapper-Klassen nicht mit Adressenvergleichen arbeiten, sondern mit equals().

Wrapper Typen verhalten sich wie Mixtur aus Standardtyp und Referenztyp.

Ähnliches Verhalten bei Strings:

String ist Klasse, damit auch Referenztyp.
Aber: Zugeständnisse an C-Programmierer:

```
String str1 = "Hallo!";
String str2 = new String("Hallo!");
String str3 = "Hallo!";
```



`str1 == str3` liefert true
`str1 == str2` liefert false

Strings sind "immutable", d.h. unveränderbar.

Regel: String Objekte mit direkter Zuweisung werden vom Compiler direkt verwaltet und verweisen bei gleichem Inhalt auf das gleiche Objekt.

Korrektur (letzte VL): 5.0/11 ≙ 0.45...45 (nicht 0.55...56)

Bei Erzeugung mit new wird immer ein neues Objekt erzeugt.

str1 = str1 + "␣Wie geht's?";
↑
neues Objekt wird erzeugt, da immutable.
str3 hat immer noch den Wert "Hallo!"

str3.equals(str2) liefert true.

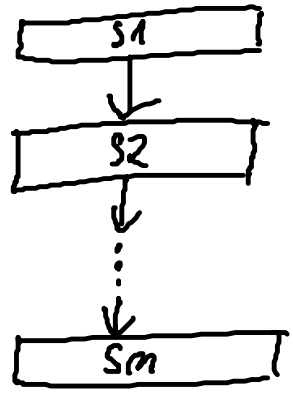
Strukturierte Anweisungen und Kontrollstrukturen

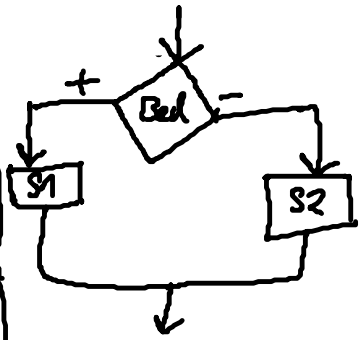
Der Programmfluss ist nicht linear „von oben nach unten", sondern kann Verzweigungen und Schleifen enthalten.

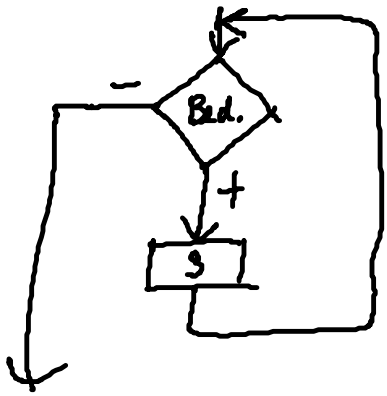
	Zusammengesetzte Anweisungen	bedingte Anweisungen	Wiederholungen

Pseudo-code	S1
	S2
	⋮
	Sn
		IF (Bed.) THEN
		S1
		ELSE
		S2
		ENDIF
			WHILE (Bed.) DO
			S
			ENDWHILE

Fluss-
diagramm







Ähnlich: Struktogramme

Diese 3 strukturellen Anweisungen reichen aus, um alles zu berechnen, was im Sinne der Theoretischen Informatik als berechenbar gilt.

Nachweis in der Informatik

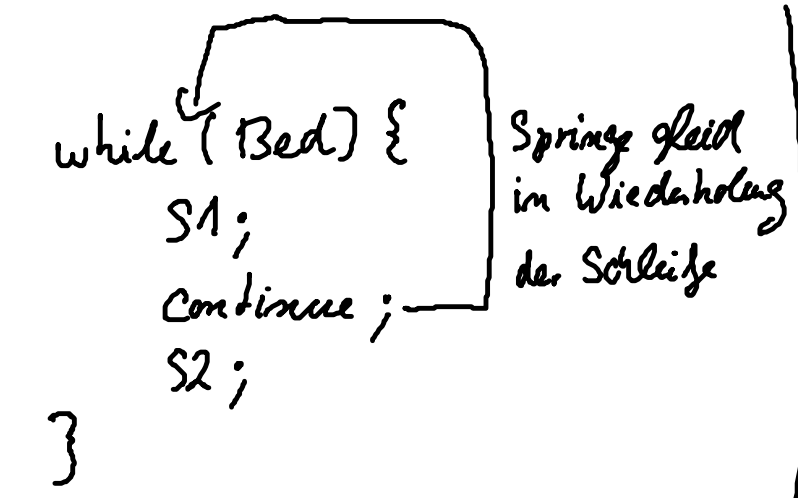
insbesondere: alle Programmiersprachen, die diese strukturellen Anweisungen kennen, sind gleich mächtig!

andere (äquivalente) Möglichkeiten existieren.

z.B.
- Zusammengesetzte Anweisungen
- bedingte Anweisungen
- goto (Sprunganweisung) VERPÖNT!

Java folgt dem mit wenigen Ausnahmen:

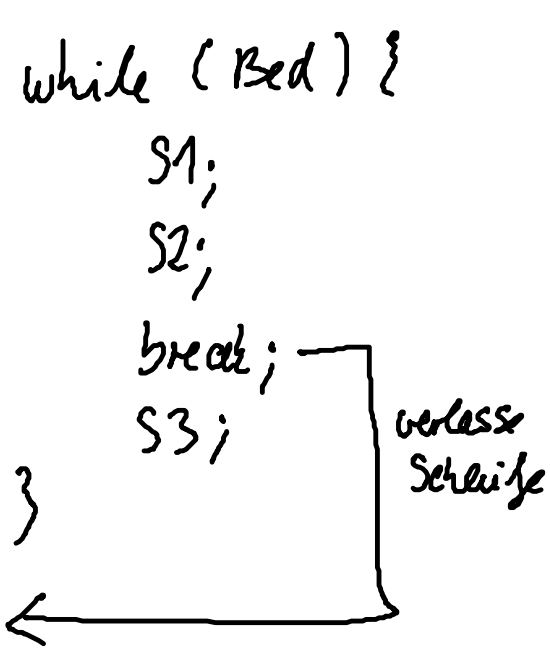
- continue
- break



Beispiel:

```
int i=0;
int sum=0;
while (i<20) {
  i++;
  if (i%2==0) {
    continue;
  } // endif
  sum += i;
} // endwhile
```

summiert ungerade Zahlen < 20



Beispiel:

```
int i=1;
while (true) {
  i++;
  if (i>10) {
    break;
  }
}
```

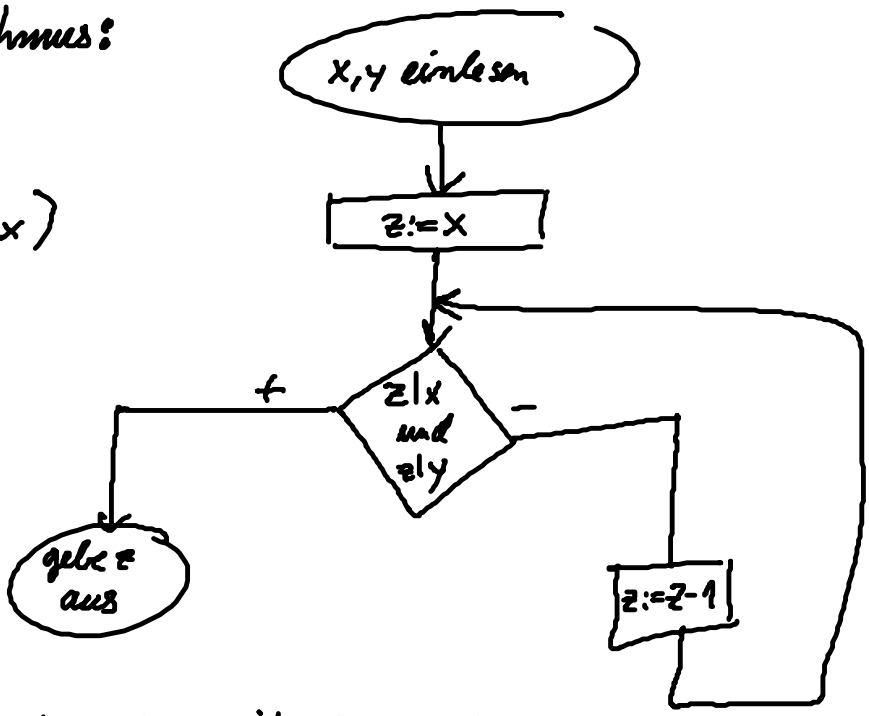
Beispiel für Strukturierung: Berechne den ggT (gcd) von zwei Zahlen $x, y \geq 1$, $x, y \in \mathbb{N}$

↑ größter gemeinsamer Teiler

Bsp: $ggT(18,27) = 9$
$ggT(12,32) = 4$

Naiver Algorithmus:
Sei $x \leq y$

($z \mid x \hat{=} z$ teilt x)



Keine Endlosschleife, da bei $z = 1$ spätestens abgebrochen wird.

Algo ist korrekt, da mit dem größtmöglichen Wert $z := x$ begonnen wird und z immer nur um eins verringert wird.

Problem: zu viele Schleifendurchläufe

Verbesserung mittels eines einfachen Lemmas:

Lemma: Seien $a > b \geq 1$, $a, b \in \mathbb{N}$. Dann gilt

$$ggT(a,b) = ggT(a-b, b)$$

Bsp: $a = 24 \longrightarrow 8 \longrightarrow 8$ $\Rightarrow ggT(24,16) = 8$
$b = 16 \qquad 16 \qquad 8$

$a = 28 \longrightarrow 16 \longrightarrow 4 \longrightarrow 4 \longrightarrow 4$ $\Rightarrow ggT(28,12) = 4$
$b = 12 \qquad 12 \qquad 12 \qquad 8 \qquad 4$

Beweis des Lemmas:

Zeige zwei Dinge:

(i) wenn k ein Teiler von a und b ist, dann ist k auch ein Teiler von $a-b$ und b.

(ii) wenn k ein Teiler von $a-b$ und b ist, dann auch von a und b.

zu (i): k Teiler von a und b

$\Rightarrow \exists\, n, m \in \mathbb{N} \setminus \{0\}$ mit $a = n \cdot k$, $b = m \cdot k$

$\Rightarrow a - b = n \cdot k - m \cdot k = \underbrace{(n-m)}_{\in \mathbb{N} \setminus \{0\}} \cdot k$

$\Rightarrow k$ ist Teiler von $a-b$ und b.

zu (ii): analog.

□