



Designing Education  
Connecting People

## Das erwartet Sie:

- OO-Konzepte
- Java
  - Logik
  - Kontrollstrukturen
  - Klassen/Objekte
  - Attribute
  - Methoden

A large graphic on the right side of the slide features a blue background with a pattern of glowing binary code (0s and 1s) and faint, larger characters, creating a digital or data-themed aesthetic.

# Funktionalität in Anwendungen realisieren



Lernfeld 11a

# Die Themen und Lernziele



## Algorithmen

---

### Lernziel

Verstehen, worum es bei Algorithmen geht und wo man sie einsetzt



## Funktionalität in Anwendungen realisieren

---

### Lernziel

Eine typsichere, objektorientierte Programmiersprache beherrschen



## Benutzerschnitt- stellen gestalten und entwickeln

---

### Lernziel

Grafische Oberflächen in einer OO-Sprache entwickeln



## Software testen

---

### Lernziel

Testfälle formulieren und anwenden

# Überblick

Java



Operatoren

Kontroll-  
strukturen

Praxis



Java

# Lernziel

Eine typsichere,  
objektorientierte  
Programmiersprache  
beherrschen



# Inkrement/Dekrement

Operator	Bezeichnung	Beispiel	Bedeutung
++	präinkrement	++c	erhöht erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
++	postinkrement	c++	führt erst die gesamte Anweisung aus und erhöht <i>danach</i> den Wert von c um 1
--	prädekrement	--c	erniedrigt erst c um 1, und führt dann den Rest der Anweisung mit dem neuen Wert aus
--	postdekrement	c--	führt erst die gesamte Anweisung aus und erniedrigt erst dann den Wert von c um 1

Wird eine Variable *c* um 1 erhöht, so kann man den Inkrementoperator *c++* oder *++c* anstatt der Ausdrücke *c = c+1* oder *c += 1* verwenden.

# Inkrement/Dekrement

```
import javax.swing.JOptionPane;

/**
 *
 * @author Student
 */
public class Inkrement {
    public static void main( String[] args ) {
        int c;
        String ausgabe = "";
        // postinkrement
        c = 5;
        ausgabe += c + ", ";
        ausgabe += c++ + ", ";
        ausgabe += c + "\n";
        // präinkrement
        c = 5;
        ausgabe += c + ", ";
        ausgabe += ++c + ", ";
        ausgabe += c;
        // Ausgabe des Ergebnisses:
        JOptionPane.showMessageDialog(null, ausgabe, "prä- und postinkrement",
        JOptionPane.PLAIN_MESSAGE);
    }
}
```

prä- und postinkrement

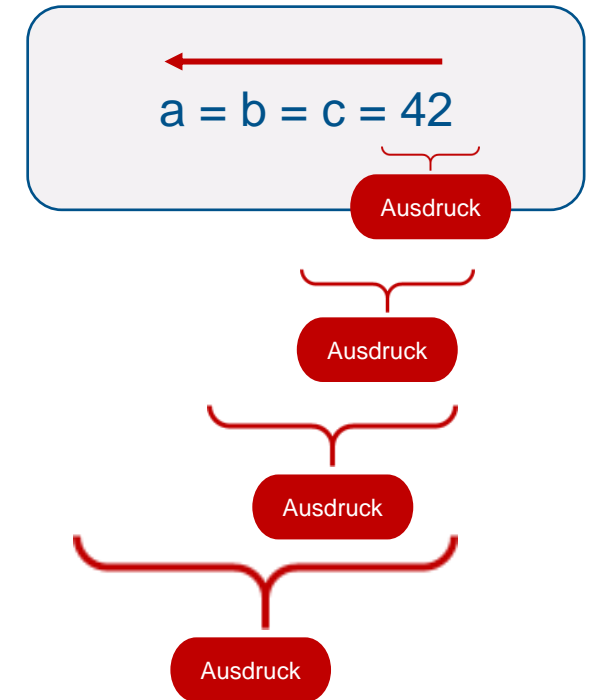
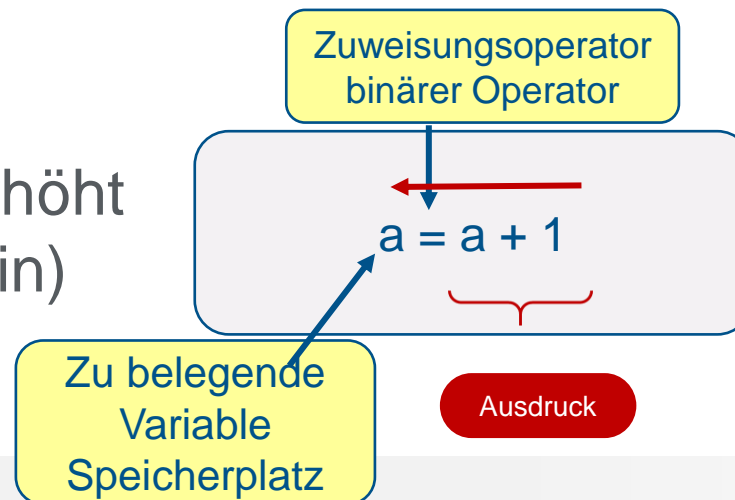
5, 5, 6

5, 6, 6

OK

# Zuweisungsoperator

- In Java ist die **Zuweisung** ein Operator
  - Sie ist ein **Ausdruck** und **keine Anweisung**
- Eine Zuweisung hat einen Rückgabewert, neben ihrem **essentiellen** „Nebeneffekt“, den Wert des linken Operanden zu verändern.  
(Zuweisungen werden von rechts nach links interpretiert !)
- Die Variable a wird um 1 erhöht  
(a kann nicht gleich a+1 sein)



# Logische Operatoren

Operator	Bezeichnung	Beispiel	Bedeutung
!	Nicht	!a b_wert = !a > b	invertiert den Ausdruck, Es wird a negiert. Ist a wahr, wird false zurückgegeben, andernfalls true.
&	UND mit vollständiger Auswertung	a & b	Es wird a und b ausgewertet. Besitzt a einen anderen Wahrheitswert als b wird true zurückgegeben, andernfalls false.
^	exklusives ODER (XOR)	a ^ b	Exor, true, wenn genau ein Operand true ist Es wird a und b ausgewertet. Besitzt a einen anderen Wahrheitswert als b wird true zurückgegeben, andernfalls false.
	ODER mit vollständiger Auswertung	a   b	Es wird a und b ausgewertet. Ist mindestens einer der beiden Ausdrücke wahr, wird true zurückgegeben, andernfalls false.
&&	UND mit kurzer Auswertung	a && b	Und, true, dann wenn beide Argumente true sind Es wird a und b ausgewertet. Die Auswertung wird abgebrochen, wenn a bereits falsch ist. Dann wird sofort false zurückgegeben. Sind beide wahr, wird true zurückgegeben, andernfalls false.
	ODER mit kurzer Auswertung	a    b	Oder, true, wenn mindestens ein Operand true ist Es wird a und b ausgewertet. Die Auswertung wird abgebrochen, wenn a bereits wahr ist. Dann wird sofort true zurückgegeben. Ist mindestens einer der beiden Ausdrücke wahr, wird true zurückgegeben, andernfalls false.

Logische Operatoren verknüpfen Wahrheitswerte miteinander.  
Logische Ausdrücke werden von links nach rechts ausgewertet.  
Die Auswertung bricht bei kurzer Auswertung ab, sobald das Ergebnis feststeht.



# Prioritäten der Vergleichsoperatoren

Operator	Bedeutung	Priorität
<	kleiner	5
<=	kleiner oder gleich	5
>	größer	5
>=	größer oder gleich	5
==	gleich	6
!=	ungleich	6

Tabelle 2.8 Vergleichsoperatoren

Der Operator mit der höchsten Priorität wird zuerst ausgewertet

== für Gleichheit  
!= für Ungleichheit

Vorsicht:  
= alleine steht für Zuweisung

Quelle: Programmieren lernen mit Java – Rheinwerk

# Prioritäten der Arithmetischen Operatoren

Operator	Bedeutung	Priorität
+	positives Vorzeichen	1
-	negatives Vorzeichen	1
++	Inkrementierung	1
--	Dekrementierung	1
*	Multiplikation	2
/	Division	2
%	Modulo (Rest)	2
+	Addition	3
-	Subtraktion	3

**Tabelle 2.9** Arithmetische Operatoren von Java

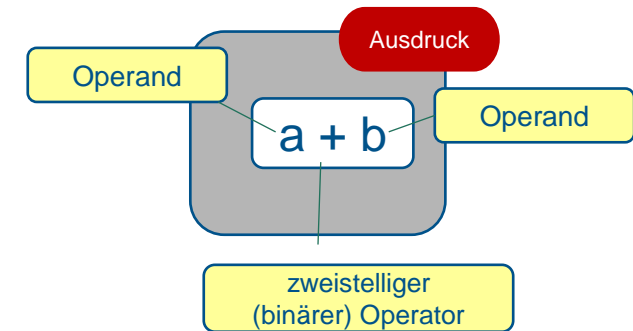
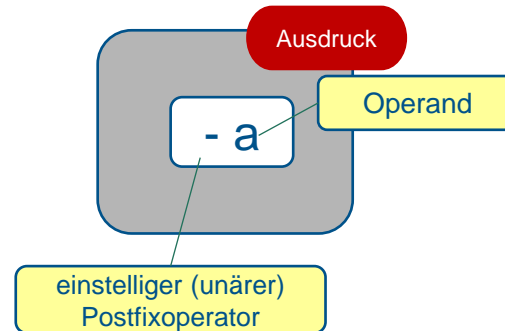
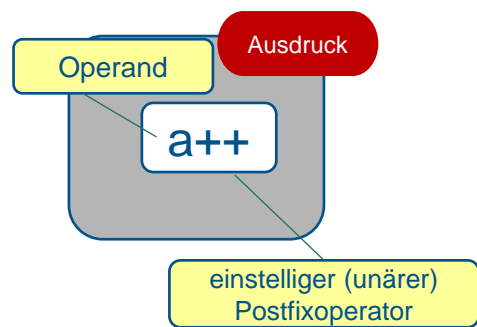
Quelle: Programmieren lernen mit Java – Rheinwerk

Der Operator mit der höchsten Priorität wird zuerst ausgewertet

# Auswertungsreihenfolge

Für Ausdrücke mit mehreren Operatoren gelten die folgenden Regeln in Bezug auf die Reihenfolge der Auswertung:

- Teilausdrücke in runden Klammern werden wie in der Mathematik als erstes ausgewertet
- Ausdrücke mit unären Operatoren werden anschließend ausgewertet
- Zuletzt werden Teilausdrücke mit mehrstelligen Operatoren ausgewertet
- Unäre Operatoren haben alle die gleiche Priorität



# Prioritäten der Logischen Operatoren

Operator	Bedeutung	Priorität
!	NICHT	1
&	UND mit vollständiger Auswertung	7
^	Exklusives ODER (XOR)	8
	ODER mit vollständiger Auswertung	9
&&	<div>S-C-E</div> UND mit kurzer Auswertung	10
	<div>S-C-E</div> ODER mit kurzer Auswertung	11

Tabelle 2.10 Logische Operatoren

Der Operator mit der höchsten Priorität wird zuerst ausgewertet

S-C-E = Short-Circuit-Evaluation, schnelles Auswerten

Quelle: Programmieren lernen mit Java – Rheinwerk

# Nicht-strikte Operatoren

- Die logischen Operatoren

`a && b` (logisches und)

`a || b` (logisches oder)

werten den zweiten Operanden nur aus, wenn wirklich benötigt

- Auswertungsabkürzung (nicht-strikte Auswertung)

- Beispiel: `if (a != 0 && b / a > 1)` [leichter zu lesen mit Klammern: `if ( (a != 0) && (b / a > 1) )`]
- Für **`a == 0`** würde **`b / a`** einen Fehler erzeugen
- Aber **`false && x`** ist immer **`false`** → `x` wird nicht ausgewertet

# Wahrheitstabelle

a	b	a & b a && b	a ^ b	a   b a    b	!a	!b
true	true	true	false	true	false	false
true	false	false	true	true	false	true
false	true	false	true	true	true	false
false	false	false	false	false	true	true



# Zusammengesetzte Boolesche Ausdrücke

```
boolean a, b; int i; char c;  
c = 'A'; i = 2; a = false;  
  
b = c == 'A' ;    // b ist jetzt true  
  
b = a && b;        // b ist jetzt false  
  
b = !b;           // b ist jetzt true  
  
b = i > 0 && 3 / i == 1;  
    // Da i == 2: b == 2 > 0 and 3 / 2  
    // Mit 3/2 == 1 (Ganzzahldivision)  
    // -> b ist jetzt true
```

# Prioritäten zusammengefasst

Precedence	Operator	Operand type	Description
1	++, --	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	( type )	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	instanceof	Object	Type comparison

# Prioritäten zusammengefasst

Precedence	Operator	Operand type	Description
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9		Integral	Bitwise OR
9		Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11		Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

Übung



# Übung

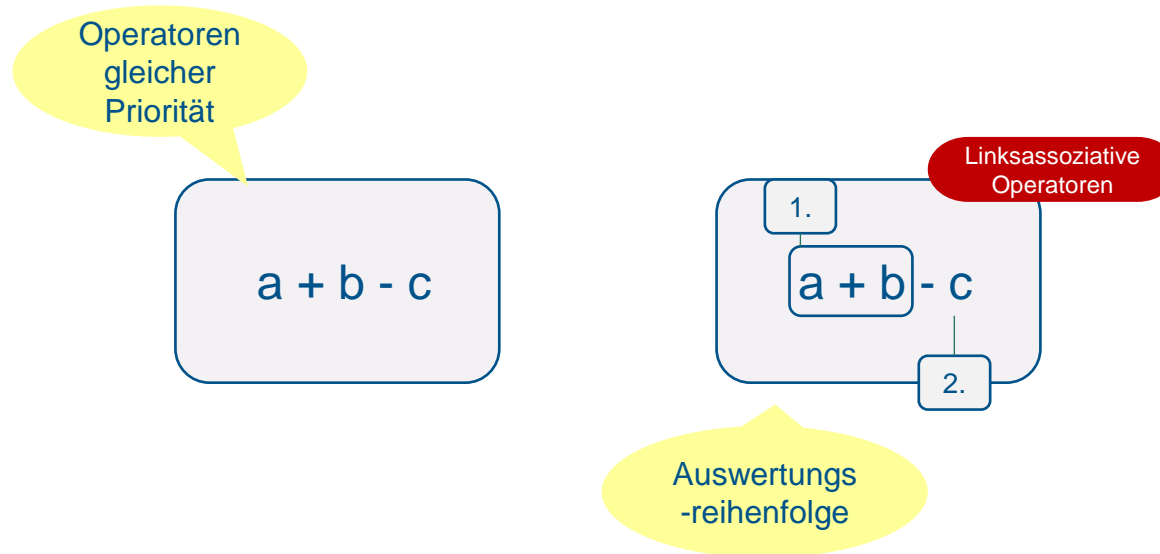
Java Programmieren

Inkrementieren/Dekrementieren, verstehen

# Auswertung von Operatoren mit gleicher Priorität

Wenn ein Ausdruck mehrere Operatoren der gleichen Priorität besitzt, wird die Auswertungsreihenfolge durch die Assoziativität der Operatoren bestimmt

Ist ein Operator linksassoziativ, wird zuerst der linke Operand ausgewertet.



# Operator-Prioritäten

Viele arithmetischen Ausdrücke können ohne Klammern geschrieben werden

- Die Auswertungsregeln entsprechen denen der Schulmathematik

```
a + b > 27 && b + c < 35 || a < 3
```

bedeutet

```
( ( (a + b) > 27) && ((b + c) < 35) ) || (a < 3)
```

Steuerung der Reihenfolge ist über Klammern veränderbar.  
Klammern erhöhen die Lesbarkeit



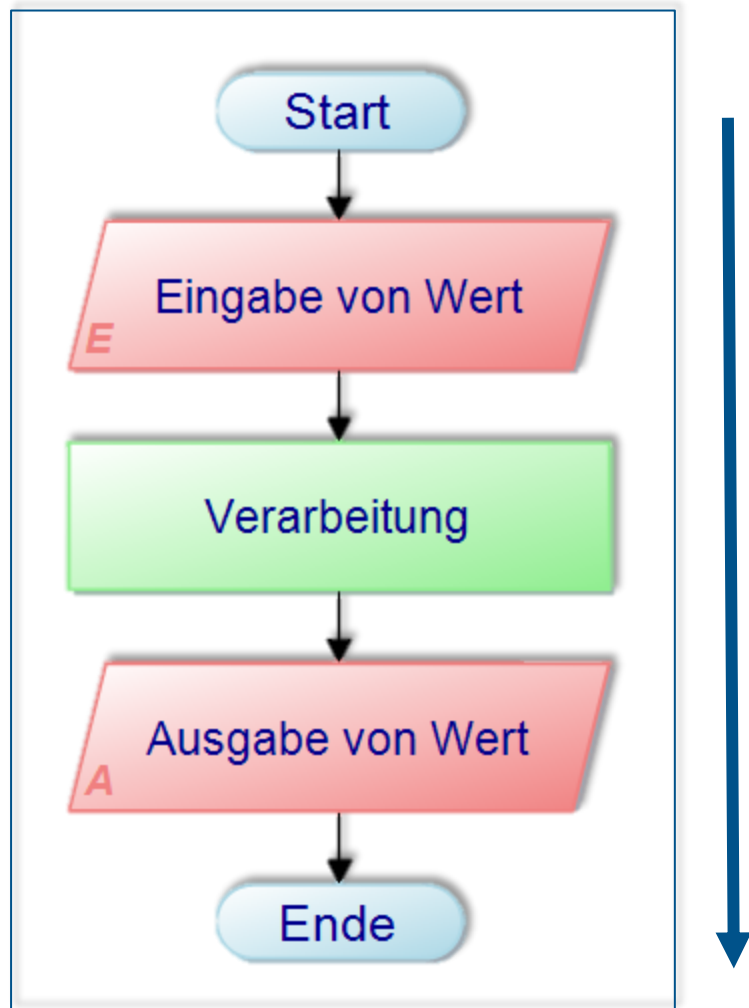
# Kompetenzcheck



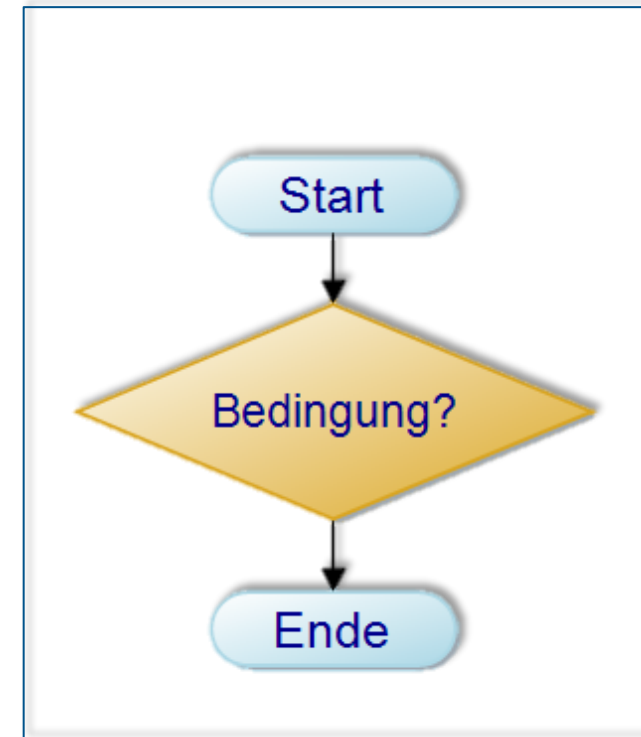
Welche Aussagen sind richtig?

- a) Logische Operatoren verknüpfen Wahrheitswerte miteinander
- b) `c++` ist dasselbe wie `c = c + 1`
- c) Das Zeichen "=" ist ein gültiger Vergleichsoperator
- d) Das logische UND wird in Java durch die Symbole "&&" dargestellt
- e) Wenn `x = True` und `Y = True` ist, was ergibt `X && Y`?
- f) Die Schlüsselwörter für eine if-Anweisung lauten `if`, `else`, `elseif`, `when`
- g) Die Reihenfolge der Auswertung ist auch über Klammern nicht veränderbar
- h) Die if-Anweisung gibt einen Wert zurück

# Kontrollstrukturen



Sequenz



Für

- IF-Anweisungen (Wenn-Dann-Sonst)
- Schleifen (Wiederholungen)

# Kontrollstrukturen

- Kontrollstrukturen beeinflussen die Ausführung von Programmen
- Zwei grundlegende Typen:
  - **Bedingung:**  
Anweisungen werden ausgeführt, wenn eine Bedingung erfüllt (**true**-Zweig) ist oder gezielt auch bei Nichterfüllung einer Bedingung (**false**-Zweig)
  - **Schleife:**  
Anweisungen werden **mehrfach ausgeführt**  
Die Kontrollanweisungen von Java werden durch folgende Grammatikregel angegeben

```
<Kontrollanweisung> =  
    <If-Anweisung> | <Switch-Anweisung> |  
    <Return-Anweisung> | <Break-Anweisung>  
    <While-Schleife> | <Do-Schleife> | <For-Schleife>
```

# Kontrollstrukturen

**if .. else**

**Verzweigung / Bedingung**

**switch .. case**

**Mehrfachverzweigung / Bedingungsstruktur**

**for**

**Zählschleife / Iteration**

**while**

**kopfgesteuerte Schleife**

**do .. while**

**fußgesteuerte Schleife**

# Fallunterscheidung: If-Anweisung

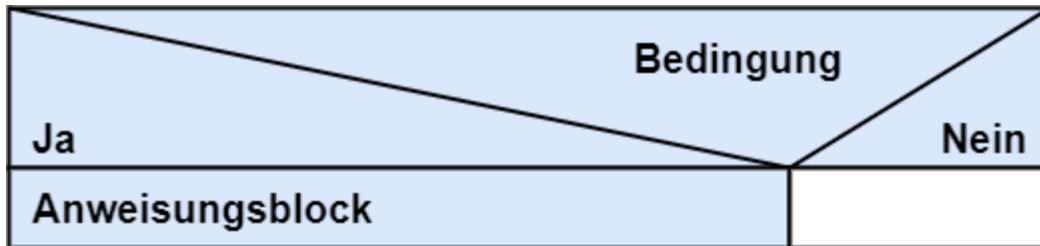
- If besteht immer aus drei Teilen:
  - Bedingung muss vom Typ *boolean* sein
  - Anweisung, wenn die Bedingung wahr (*true*) ist
  - Anweisung, wenn die Bedingung falsch (*false*) ist
- Anweisung kann ein Block sein, also eine Sequenz von Ausdrücken
- *if*-Anweisungen geben keinen Wert zurück
- Der *else*-Zweig ist optional, ausgedrückt durch [ ]

```
<If-Anweisung> =  
    if (<Ausdruck>)  
        <Anweisung>  
    [else  
        <Anweisung>]
```

# Verzweigungen (Selektion)

## Einseitige Verzweigungen

Ein Anweisungsblock wird nur unter bestimmten Bedingungen durchlaufen



```
if (Bedingung) {  
    Anweisung1  
    Anweisung2  
}
```

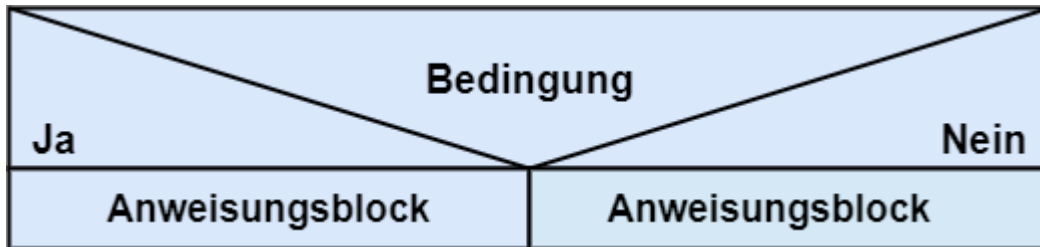
```
float zahl= scanner.nextFloat() ;  
if (zahl >= 10.0 && zahl <=20.0) {  
    System.out.println("Die Zahl " + zahl + "liegt im Bereich von 10 bis 20");  
}
```



# Verzweigungen (Selektion)

## Zweiseitige Verzweigungen

Je nach Bedingung wird einer der zwei Anweisungsblöcke durchlaufen



```
if (Bedingung) {  
    Anweisung1  
    ...  
}  
else {  
    Anweisung2  
    ...  
}
```

```
float zahl= scanner.nextFloat() ;  
if (zahl >= 10.0 && zahl <=20.0) {  
    System.out.println("Die Zahl " + zahl + "liegt im Bereich von 10 bis 20");  
}  
else {  
    System.out.println("Die Zahl " + zahl + " liegt nicht im Bereich von 10 bis 20");  
}
```

# Verzweigungen (Konditionaloperator)

## Bedingungsoperator/Konditionaloperator

Der Wert eines Ausdrucks ist von einer Bedingung abhängig, ohne eine if-Anweisung zu verwenden

**Bedingung ? Ausdruck, wenn wahr : Ausdruck, wenn falsch ;**

```
// ermitteln des Maximums
int a = 3, b = 5, max;
max = (a > b) ? a : b;
System.out.println("Der größte Wert ist: " + max);
```

entspricht folgender If-Anweisung:

```
// ermitteln des Maximums
if (a > b)
    max = a;
else
    max = b;
```

# Kontrollstrukturen

if .. else

Verzweigung / Bedingung

switch .. case

Mehrfachverzweigung / Bedingungsstruktur

for

Zählschleife / Iteration

while

kopfgesteuerte Schleife

do .. while

fußgesteuerte Schleife

# Fallunterscheidung: Switch-Anweisung

- Die Switch–Anweisung hat mehrere Sprungziele
- Je nach Wert soll eine bestimmte Anweisung erfolgen
- Sie erlaubt die Auswahl von: Ganzzahlen, Wrapper-Typen, Aufzählungen und Strings
- Der **default**-Zweig wird durchlaufen, wenn kein **case** zutrifft, ist optional, maximal **eine default**-Anweisung am Ende der Switch-Anweisung

```
<Switch-Anweisung> =  
    case (<Ausdruck>) :<Anweisung>  
    case (<Ausdruck>) :<Anweisung>  
    case (<Ausdruck>) :<Anweisung>  
    :  
    [default:  <Anweisung>]
```

# Verzweigungen (Selektion)

## Fallunterscheidung

abhängig vom Wert einer Variablen (Selektor) werden verschiedene Anweisungsblöcke durchlaufen

Selektor			
Wert 1	Wert 2	Wert 3	Sonst
Anweisungsblock 1	Anweisungsblock 2	Anweisungsblock 3	Anweisungsblock 4

```
int auswahl= scanner.nextInt();
switch (auswahl){
    case 1:
        System.out.println("Es wurde eins eingegeben");
    case 2:
        System.out.println("Es wurde zwei eingegeben");
    case 3:
        System.out.println("Es wurde drei eingegeben");
    default:
        System.out.println("Es wurde keine 1,2 oder 3 eingegeben");
}
```


```
switch (Selektor){
case Wert1:
    Anweisungsblock1
case Wert2:
    Anweisungsblock2
case Wert3:
    Anweisungsblock3
default:
    Anweisungsblock4
}
```

# Verzweigungen (Selektion)

## Fallunterscheidung

Trifft kein case (Fall) zu, wird der default-Zweig durchlaufen

```
// Switch ohne Break --> "Durchfall"
int farbeingabe = 1;
switch (farbeingabe){
    case 0: System.out.println("rot");
    case 1: System.out.println("grün");
    case 2: System.out.println("blau");
    default: System.out.println("Farbe nicht identifizierbar");
}
```



```
run:
grün
blau
Farbe nicht identifizierbar
```

Nicht das, was  
wir wollten

```
switch (Selektor){
    case Wert1:
        Anweisungsblock1;
    case Wert2:
        Anweisungsblock2;
    case Wert3:
        Anweisungsblock3;
    default:
        Anweisungsblock4;
}
```

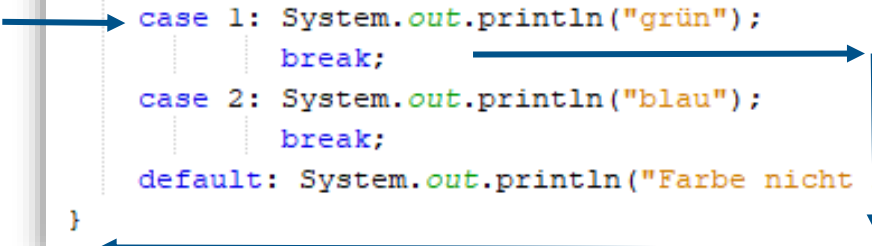


# Verzweigungen (Selektion)

## Fallunterscheidung

- **break**-Anweisung kann verwendet werden, um den **switch**-Block zu verlassen
- Ist die letzte Anweisung in einem **case**-Block

```
// Switch ohne Break --> "Durchfall"
int farbeingabe = 1;
switch (farbeingabe){
    case 0: System.out.println("rot");
            break;
    case 1: System.out.println("grün");
            break;
    case 2: System.out.println("blau");
            break;
    default: System.out.println("Farbe nicht identifizierbar")
}
```



```
run:
grün
```

```
switch (Selektor){
case Wert1:
    Anweisungsblock1; break;
case Wert2:
    Anweisungsblock2; break;
case Wert3:
    Anweisungsblock3; break;
default:
    Anweisungsblock4
}
```

# Kontrollstrukturen

if .. else	Verzweigung / Bedingung
switch .. case	Mehrfachverzweigung / Bedingungsstruktur
<b>for</b>	<b>Zählschleife / Iteration</b>
while	kopfgesteuerte Schleife
do .. while	fußgesteuerte Schleife

# Schleifen (Iterationen)

## Zählschleife

Die einfachste Schleife wiederholt ihre Anweisung eine feste Anzahl mal („counting loop“)

```
<For-Schleife> =  
    for ( [<Anweisung | <Variablen-Deklaration>];  
          [<Ausdruck>]; [<Anweisung>] )  
        <Anweisung>
```

Ausdruck muss vom Typ **boolean** sein

# Schleifen (Iterationen)

## Zählschleife

- Es kann sowohl „vorwärts“ als auch „rückwärts“ wiederholt werden
- Die Schrittweite kann festgelegt werden

```
// vorwärts
System.out.println("\nVorwärts");
int start= 0, end = 10;
for (int j = start; j < end; j++){
    System.out.print(j + " ");
}
// rückwärts
System.out.println("\nRückwärts");
for (int j = end; j > start; j-=2){
    System.out.print(j + " ");
}
```

Schrittweite 1



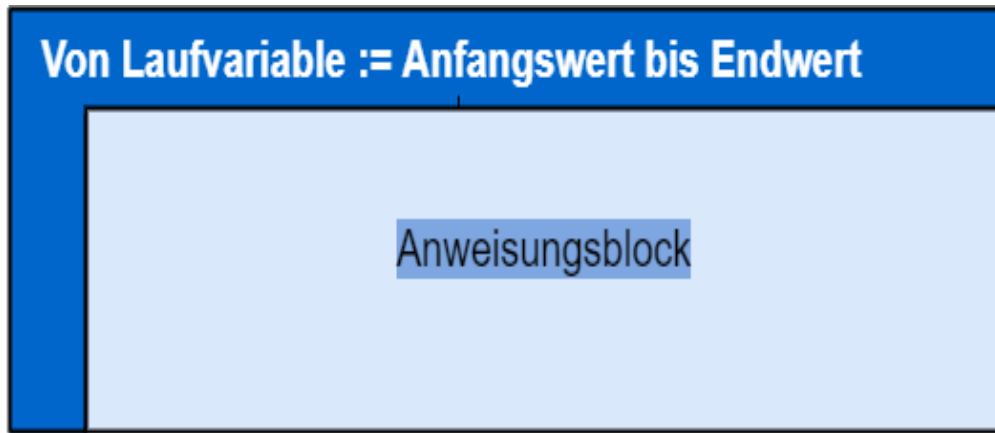
Schrittweite 2

```
Vorwärts
0 1 2 3 4 5 6 7 8 9
Rückwärts
10 8 6 4 2
```

# Schleifen (Iterationen)

## Zählschleife

sinnvoll bei bekanntem Anfangswert und Anzahl der Wiederholungen



```
for (Initialisierung; Bedingung; In-/Dekrementierung) {  
    Anweisung1  
    Anweisung2  
    ...  
}
```

```
int i;  
for (i=0; i<10; i++)  
    System.out.print(" * ");
```



```
run:  
* * * * *
```

# Schleifen

```
<label:>
for (Initialisierung; Bedingung; In-/Dekrementierung) {
    Anweisung1
    Anweisung2
    ...
    [break <label>;]
    [continue <label>;]
}
```

Beendet die aktuelle Schleife

Unterbricht die aktuelle Schleife und springt an die Wiederholungsbedingung

Weiterentwickelte for-Schleife:

```
for (Datentyp d : collection) {
    ...;
}
```

# Kontrollstrukturen

if .. else	Verzweigung / Bedingung
switch .. case	Mehrfachverzweigung / Bedingungsstruktur
for	Zählschleife / Iteration
<b>while</b>	<b>kopfgesteuerte Schleife</b>
do .. while	fußgesteuerte Schleife

# Schleifen (Iterationen)

## Kopfgesteuerte Schleife

- Häufig werden Schleifen nicht eine feste Anzahl mal ausgeführt
- Sie führt eine Anweisung aus, **solange** die Schleifen-Bedingung erfüllt (**true**) ist

```
<While-Schleife> =  
    while (<Ausdruck>)  
        <Anweisung>
```

Ausdruck muss vom Typ **boolean** sein

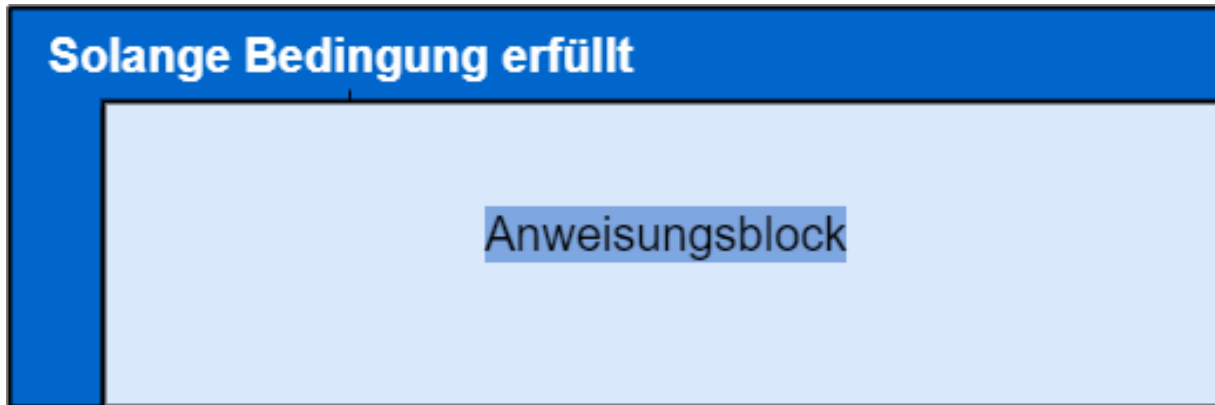
- **Vor der ersten** und **vor jeder weiteren** Ausführung der Anweisung wird die Schleifenbedingung geprüft
  - Sobald die Bedingung als **false** ausgewertet wird, **endet** die Schleife
- Es ist nicht sicher, dass der Schleifenkörper überhaupt ausgeführt wird



# Schleifen (Iterationen)

## Kopfgesteuerte Schleife

Die Bedingung für die Wiederholung steht am Anfang (Kopf)



```
while (Bedingung) {  
    Anweisung1  
    Anweisung2  
    ...  
}
```

```
//Kopfgesteuerte Schleife  
int k = 0;  
while (k <=10){  
    System.out.println(k);  
    k = k + 1;  
}
```



```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

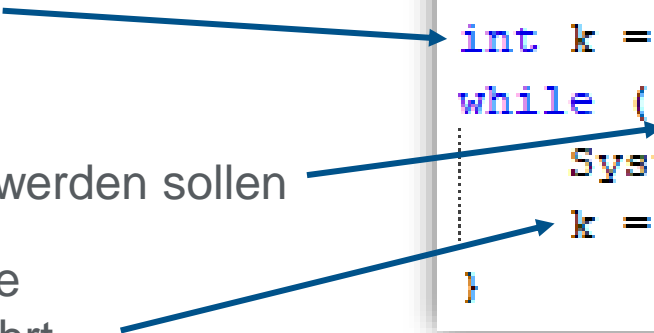
# Schleifen (Iterationen)

## Kopfgesteuerte Schleife

### Vorgehen:

- **Initialisierung** von Variablen, die für die Bedingung verwendet wird/werden
- **Bedingung**, unter der die Anweisungen ausführt werden sollen
- Die Anweisungen für den Schleifenkörper und die **Aktion**, die **näher** an den **Abbruch** der Schleife führt
- Auch diese Schleife kann mit **break** beendet oder mit **continue** unterbrochen werden

```
//Kopfgesteuerte Schleife
int k = 0;
while (k <=10){
    System.out.println(k);
    k = k + 1;
}
```



# Kontrollstrukturen

if .. else	Verzweigung / Bedingung
switch .. case	Mehrfachverzweigung / Bedingungsstruktur
for	Zählschleife / Iteration
while	kopfgesteuerte Schleife
<b>do .. while</b>	<b>fußgesteuerte Schleife</b>



# Übung

## Java Programmieren

Zahlendreieck – mit For- und While-Schleife

# Schleifen (Iterationen)

## Fußgesteuerte Schleife

- Sie führt eine Anweisung aus, **solange** die Schleifen-Bedingung erfüllt (**true**) ist

```
<Do-While-Schleife> =  
    do  
        <Anweisung>  
    while (<Ausdruck>)
```

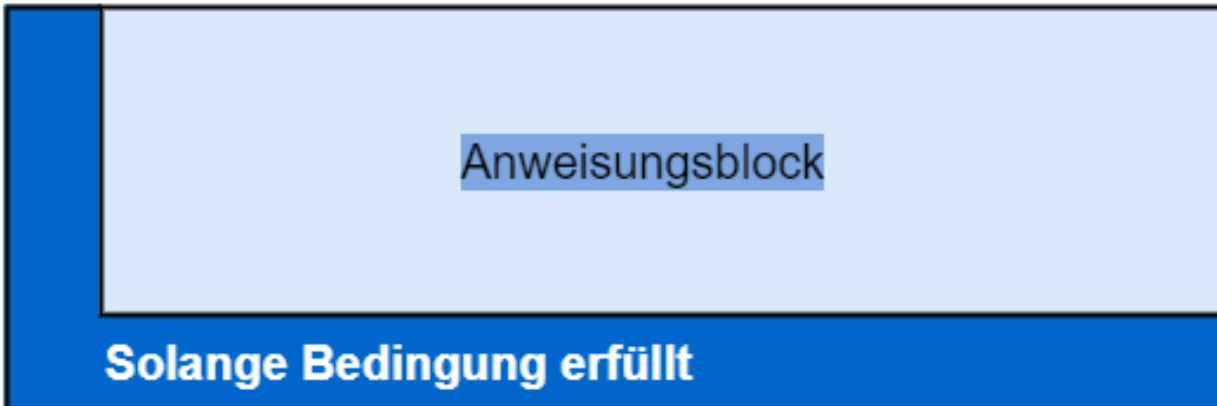
Ausdruck muss vom Typ **boolean** sein

- Nach der ersten und nach jeder weiteren Ausführung der Anweisung wird die Schleifenbedingung geprüft
  - Sobald die Bedingung als **false** ausgewertet wird, endet die Schleife
- Es ist sicher, dass der Schleifenkörper mindestens einmal ausgeführt wird

# Schleifen (Iterationen)

## Fußgesteuerte Schleife

- Die Bedingung für die Wiederholung steht am Ende (Fuß)
- Der Anweisungsblock wird mindestens einmal durchlaufen
- Deshalb auch bezeichnet als "nicht abweisende Schleife"



```
do {  
    Anweisung1  
    Anweisung2  
    ...  
} while (Bedingung) ;
```

# Schleifen (Iterationen)

## Fußgesteuerte Schleife

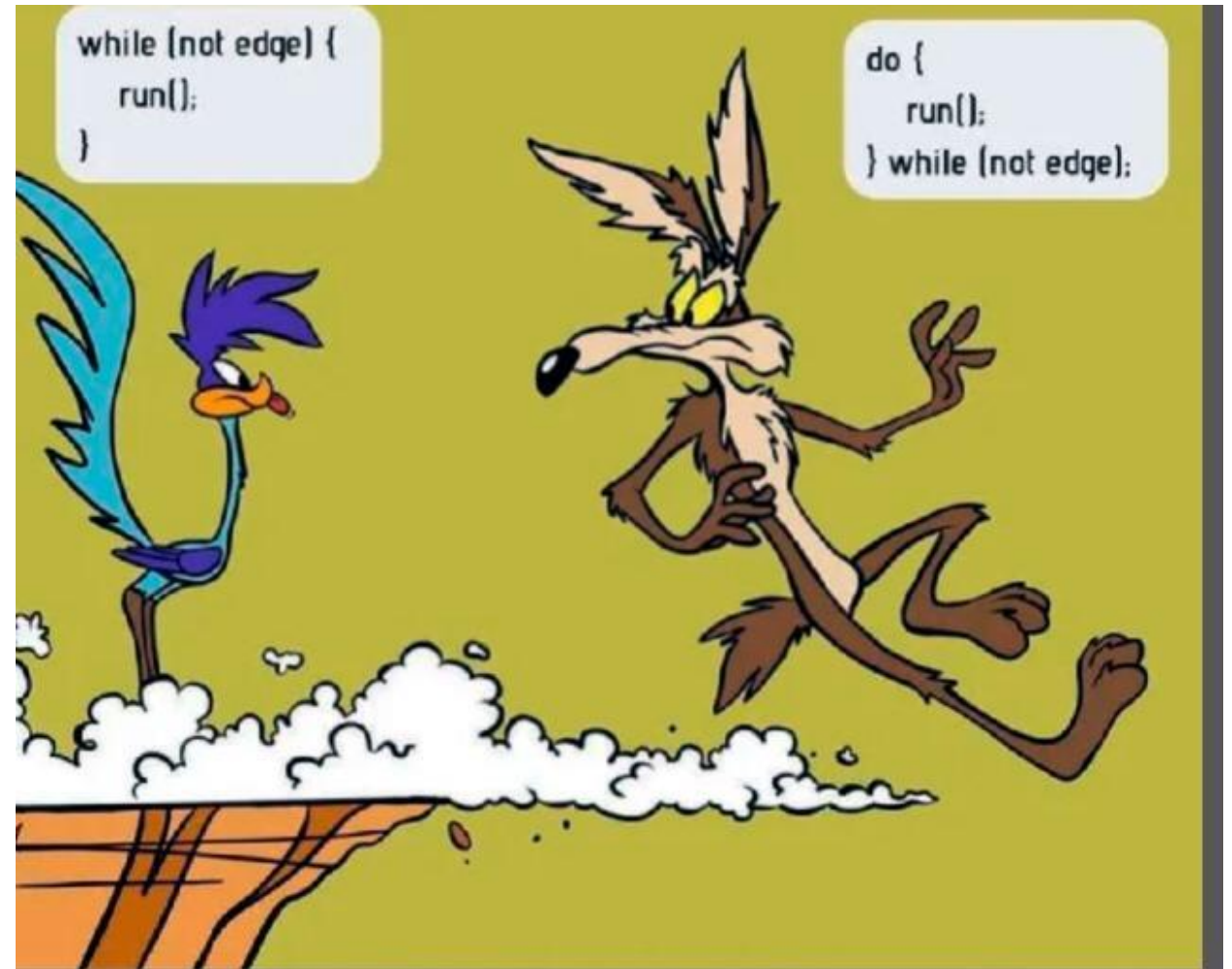
```
// Fußgesteuerte Schleife mit Konsoleneingabe
Scanner input = new Scanner(System.in);
String text="";
do {
    System.out.println("Bitte geben Sie einen Text ein: ");
    text = input.next();
    System.out.println("text.length() " + text.length());

    if (text.length() == 0 || text.equals("q")){
        break;
    }
    else
        System.out.println("Eingegebener Text: " + text);
}
while (true);
```

Endlos-Schleife?  
Was tun?

# Schleifen (Iterationen)

- Was ist wann sinnvoll?
- Kopfgesteuert
- Fußgesteuert
- Diskutieren Sie und finden sinnvolle Beispiele





# Schleifen (Iterationen) – While oder For?

## While

---

```
<initialization>;  
while (<condition>) {  
    <core loop body>;  
    <loop advancement>;  
}
```

## For

---

```
for (<initialization>; <condition>; <loop advancement>)  
    <core loop body>
```

# Kompetenzcheck



Welche Aussagen sind richtig?

- a) Es gibt nur die zweiseitige Verzweigung
- b) In der Switch-Anweisung kann es beliebig viele default-Zweige geben
- c) Schleifen werden immer einmal durchlaufen
- d) Die fußgesteuerte Schleife beginnt in Java mit dem Schlüsselwort "*repeat*",
- e) Die Schlüsselwörter für eine kopfgesteuerte Schleife sind „*do*“ und „*while*“
- f) Kopfgesteuerte Schleifen werden nicht durchlaufen, wenn die Schleifenbedingung schon vorher nicht erfüllt ist
- g) Schleifen (zähl-, kopfgesteuerte -, fußgesteuerte -) können gemischt werden

Übung



# Übung

Java Programmieren

Performancetest „GGT“

Übung



# Übung

Java Programmieren

Kapitalverdoppelung

Übung



# Übung

Java Programmieren

Collatz-Zahlenfolge