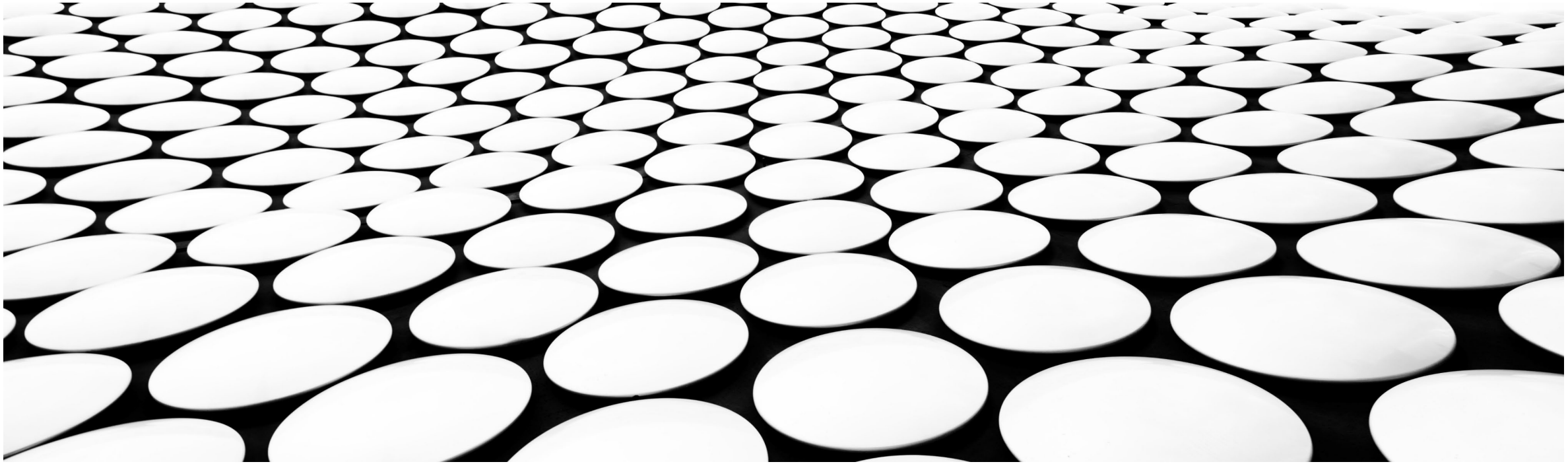
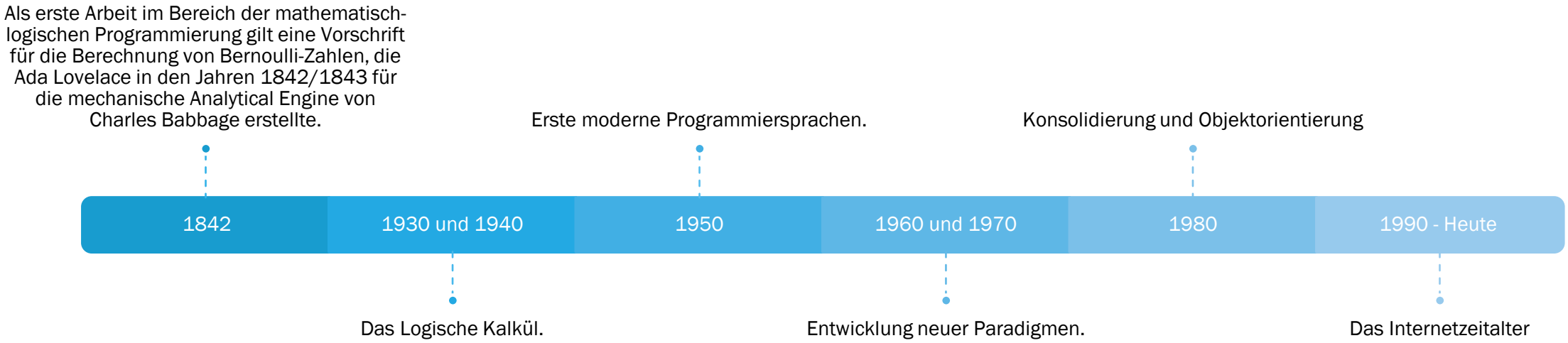
GRUNDLAGEN PROGRAMMIEREN

MIT BEISPIELEN IN JAVA



GESCHICHTE DER PROGRAMMIERSPRACHEN



Mehr gibt es auf
https://de.wikipedia.org/wiki/Geschichte_der_Programmiersprachen

BIT FÜR BIT ERGIBT SICH EIN BYTE

VON DER 0 ZUR 1 SOWIE DIE DARSTELLUNG VON 2 ODER 8 ODER 1024

BIT -> BYTE -> SHORT -> INTEGER -> LONG

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

- Die Bitfolge stellt eine 16 Bit lange Zahl dar.
- Alle gesetzten Bits werden aufaddiert.
- Für den Computer ergibt die obige Bitfolge:
  0*32768 + 0*16384 + 0*8192 + 0*4096 + 0*2048 + 0*1024 + 0*512 + 0*256 + 0*128 +
  0*64 + 0*32 + 0*16 + 0*8 + 0*4 + 0*2 + 0*1
- Bzw.
  0+0+0+0+0+0+0+0 = 0
- Welcher ist der Größte darzustellende Wert?

SIGNED UND UNSIGNED / MIT UND OHNE VORZEICHEN NEGATIVE UND POSITIVE ZAHLEN

32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Die Bitfolge stellt eine 16 Bit lange Zahl dar.
Es gibt Programmiersprachen die Zwischen „**Signed**“ und „**Unsigned**“ Datentypen unterscheiden.

- **Signed** – Bedeutet Vorzeichenbehaftet
  - Ein **Signed** Datentyp kann Positive wie Negative Werte aufnehmen
  - Bei **Signed** Datentyp wird das höherwertige Bit für das Vorzeichen verwendet.
    - Dieses Bit gilt als Marker Bit. Alle andere gesetzten Bits werden vom Wert des Marker Bits abgezogen.
  - Zero, also 0 ist auch ein Zustand. (Immer dran denken)
- **Unsigned** – Bedeutet ohne Vorzeichen
  - Ein **Unsigned** Datentyp kann nur Positive Werte aufnehmen
  - Hier werden alle Bits für den Wertebereich verwendet.
    - Welche ist die Größte darstellbar Ziffer?

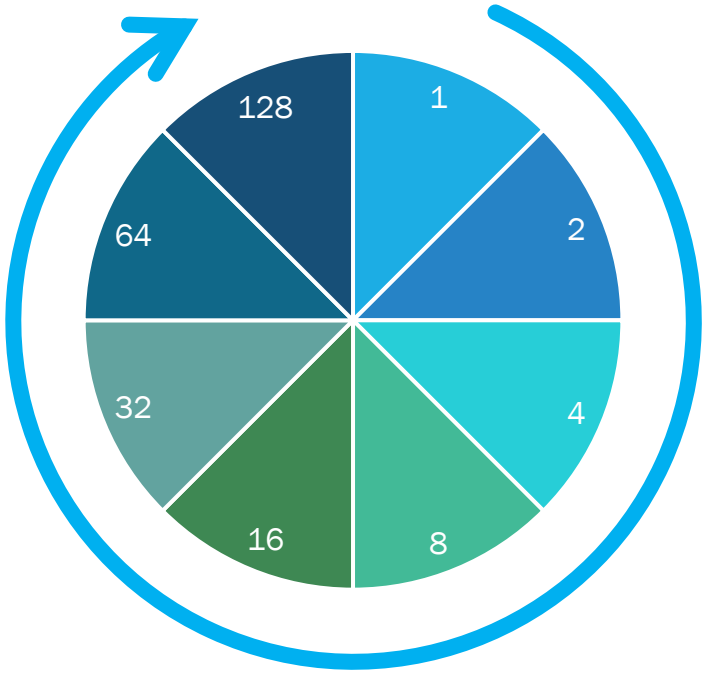
BIT -> BYTE -> SHORT -> INTEGER -> LONG

Value	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
32719	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1

- Das Grün dargestellte Bit ist das Marker Bit.
- Ist dieses alleine gesetzt ist der Wert negiert 32.768 also -32.768
- Werden weitere Bits gesetzt werden diese von dem Wert 32.768 abgezogen.
  - Abgezogen weil alle anderen Bits Positive Zahlen repräsentieren.
  - -32.768 + 1 = 32.767
  - -32.768 + 1*32 + 1*16 + 1*1 = ???
  - -32.768 + 1*32 + 1*16 + 1*1 = 32.719

WARUM IST 127 + 1 NICHT 128?

Die Bit Uhr

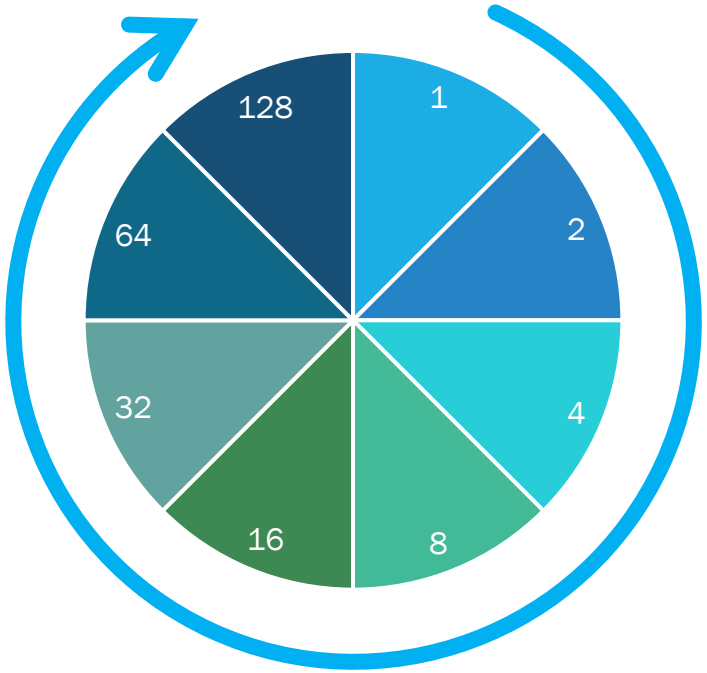


Signed Datentype nutzen das Höchste Bit für das Vorzeichen.

Ziffer	128	64	32	16	8	4	2	1
?	0	0	0	0	0	0	1	1
?	0	1	1	1	1	1	1	1
?	1	1	1	1	1	1	1	1
?	1	0	1	0	1	0	1	0
-127	1	0	0	0	0	0	0	1

127 + 1 IST JA DOCH 128
ABER 255 + 1 IST 0

Die Bit Uhr



Unsigned Datentype nutzen alle Bits für den Wertebereich

Ziffer	128	64	32	16	8	4	2	1
?	0	0	0	0	0	0	1	1
?	0	1	1	1	1	1	1	1
?	1	1	1	1	1	1	1	1
?	1	0	1	0	1	0	1	0
?	0	1	0	1	0	1	0	1
?	1	1	1	1	1	1	1	1

ARITHMETISCHE OPERATOREN

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel	Art

Inkrement;
Erhöht eine Variable um den Wert 1
	++	Ganze Zahlen,
		Fließkommazahlen
			int a = 3;
			a++;
			float b = 3f;
			b++;
				Unär

Arithmetische Addition;
Das Ergebnis der Berechnung entsprich dem des Operanden mit dem größten Wertebereich
z.B. int+int = int
int + long = long
	+	Ganze Zahlen,
		Fließkommazahlen
			int c,d,e;
			c = 3;
			d = 5;
			e = c + d;
			
			int x;
			long y,z;
			x = 3;
			y = 4;
			z = x + y;
				Binär

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel	Art

Arithmetische Subtraktion	-	Ganze Zahlen, Fließkommazahlen	int e = 3;
			float f = 4;
			float g;
			g = e – f;
				Binär

Arithmetische Multiplikation	*	Ganze Zahlen, Fließkommazahlen	int c,d;
			c = 3;
			d = c * 4;
				Binär

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel	Art

Arithmetische Division; berechnet Quotienten aus Dividend und Divisor	/	Ganze Zahlen: Sind beide Operanden ganze Zahlen, wird das Ergebnis nach dem Komma abgeschnitten
		
		Fließkommazahlen: Ist mindestens ein Operand eine Fließkommazahl, wird das Ergebnis auf eine Fließkommazahl und nicht gerundet.
			int e = 3;
			float f = 4;
			float g;
			g = e / f;
				Binär

Rest (auch: Restwert Operator – Modulo genannt);
Berechnet den Rest der arithmetischen Division
	%	Ganze Zahlen, Fließkommazahlen	int k =11;
			int l = 5;
			int m;
			m = k % l;
				Binär

LOGISCHE OPERATOREN

WERTEN AUSDRÜCKE ZU WAHR (TRUE) ODER FALSCH (FALSE) AUS

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Logische Komplement (Negation); Ändert der Wahrheitswert des Operanden	!	boolean	boolean b2 = false; boolean b1 = !b2;

Logisches UND;
Liefert true, wenn beide Operanden true sind.
	&&	boolean	boolean b4 = true;
			boolean b5 = true
			boolean b3 = b4 && b5;

Logische Oder;
Liefert true, wenn einer der beiden Operanden true ist
	||	boolean	boolean b7 = false;
			boolean b8 = true
			boolean b6 = b7 || b8;

Exclusiv-Oder;
Liefert true, wenn nur einer der beiden Operanden true ist
	^	boolean	boolean b10 = false;
			boolean b11 = true
			boolean b9 = b10 ^ b11;

VERGLEICHSOPERATOREN

VERGLEICHEN AUSDRÜCKE - LIEFERN WAHR (TRUE) ODER FALSCH (FALSE)

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel

Gleichheit;
Primitive Datentypen:
Liefert true, wenn die Werte der Operanden gleich sind
	==	Primitive Datentypen	int z1, z2;
			boolean e1;
			z1 = 3;
			z2 = 3;
			e1 = z1 == z2;

Gleichheit;
Referenzdatentypen
Liefert true, wenn in beiden Opreanden die Refenrenz auf dasselbe Objekt enthalten ist
	==	Referenzdatentypen	Kunde kunde1, kunde2;
			boolean e2;
			kunde1 = new Kunde();
			kunde2 = kunde1;
			e2 = kunde1 == kunde2;

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel

Ungleichheit;
Primitive Datentypen:
Liefert true, wenn die Werte der Operanden nicht gleich sind
	!=	Primitive Datentypen	int z3, z4;
			boolean e3;
			z1 = 4;
			z2 = 3;
			e3 = z3 != z4;

Ungleichheit;
Referenzdatentypen
Liefert true, wenn in beiden Operanden nicht die Refenrenz auf dasselbe Objekt enthalten ist
	!=	Referenzdatentypen	Kunde kunde3, kunde4;
			boolean e4;
			kunde3 = new Kunde();
			kunde4 = new Kunde();
			e4 = kunde1 != kunde2;

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel

Kleiner als
liefert true, wenn der Wert des Linken Operanden kleiner ist verglichen mit dem Wert des rechten Operanden
	<	Ganz Zahlen, Fließkommazahlen	int z5, z6;
			boolean e5;
			z5 = 4;
			z6 = 5;
			e5 = z5 < z6;

Kleiner gleich
liefert true, wenn der Wert des Linken Operanden kleiner oder gleich ist verglichen mit dem Wert des rechten Operanden
	<=	Ganz Zahlen, Fließkommazahlen	int z5, z6;
			boolean e5;
			z5 = 4;
			z6 = 5;
			e5 = z5 <= z6;

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel

Größer als
liefert true, wenn der Wert des Linken Operanden größer ist verglichen mit dem Wert des rechten Operanden
	>	Ganz Zahlen, Fließkommazahlen	int z7, z8;
			boolean e6;
			z7 = 6;
			z8 = 5;
			e6 = z7 > z8;

Größer gleich
liefert true, wenn der Wert des Linken Operanden größer oder gleich ist verglichen mit dem Wert des rechten Operanden
	>=	Ganz Zahlen, Fließkommazahlen	int z7, z8;
			boolean e6;
			z7 = 6;
			z8 = 5;
			e6 = z7 >= z8;

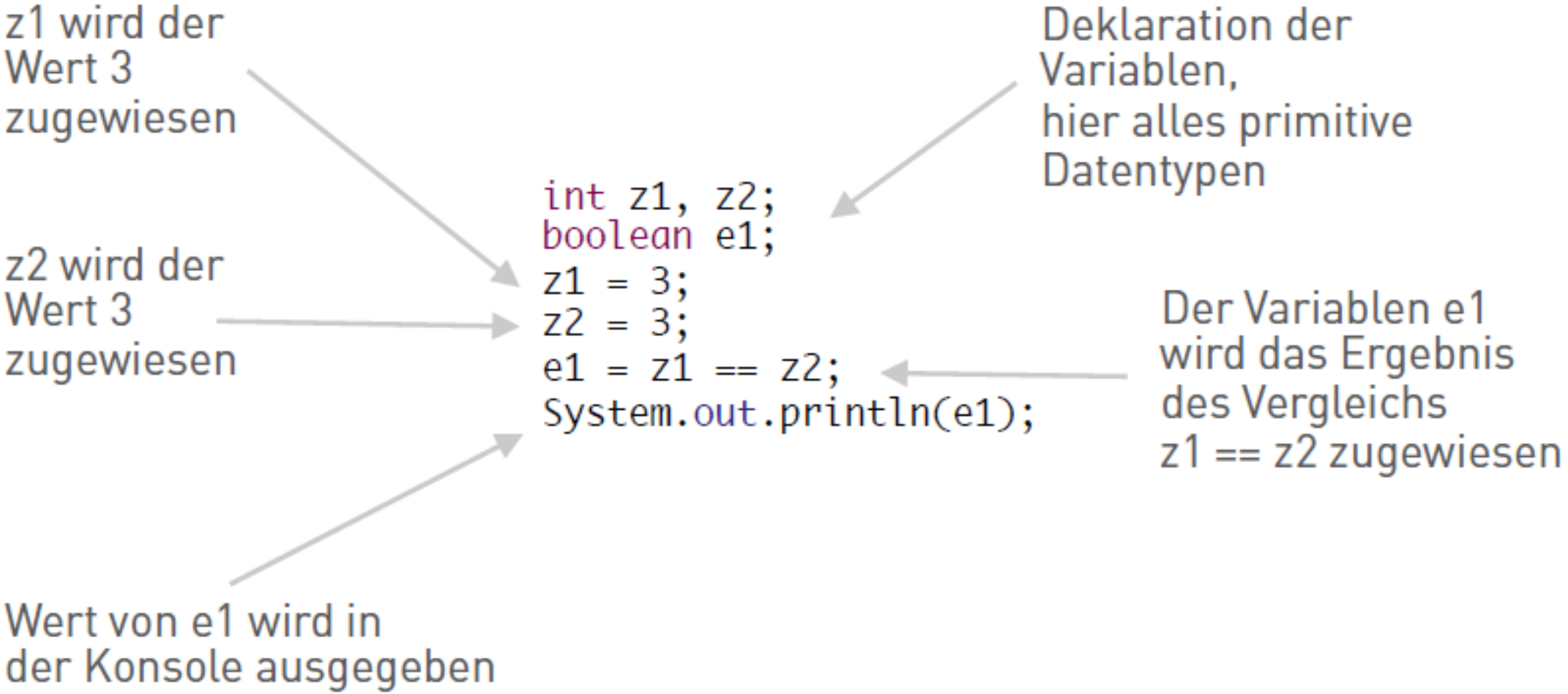
OPERATOREN UND AUSDRÜCKE

- Besondere Prüfoperatoren
  - Prüfung auf Gleichheit (==)
  - Prüfung auf Ungleichheit (!=)
- Unterschied beim Vergleich zwischen primitiven Datentypen und Referenzdatentypen
  - bei primitiven Datentypen werden Werte verglichen
  - Bei Referenzdatentypen werden Referenzen verglichen

VERGLEICH PRIMITIVER DATENTYPEN

INT, LONG, FLOAT, DOUBLE, SHORT, BYTE...

OPERATOREN UND AUSDRÜCKE



VERGLEICH VON REFERENZDATENTYPEN

OPERATOREN UND AUSDRÜCKE

```
Kunde k3, k4;

boolean e8;

k3 = new Kunde();

k4 = new Kunde();

e8 = k3 == k4;

System.out.println(e8);
```

Deklaration der Variablen k3 und k4, Datentyp ist Kunde, d.h. kein primitiver Datentyp sondern ein Referenzdatentyp

k3 wird die Referenz auf ein neu erzeugtes Objekt vom Typ Kunde zugewiesen

k4 wird die Referenz auf ein neu erzeugtes Objekt vom Typ Kunde zugewiesen

Der Variablen e8 wird das Ergebnis des Vergleichs k3 == k4 zugewiesen

Wert von e8 wird in der Konsole ausgegeben

OPERATOREN UND AUSDRÜCKE

- k3 und k4 wird ein neu erzeugtes Objekt zugewiesen
  - In k3 ist die Referenz auf das erste Objekt „Kunde“ gespeichert
  - in k4 die Referenz auf das zweite Objekt „Kunde“
- Der Vergleich auf „Referenzgleichheit“ von k3 und k4 liefert damit den Wert false

OPERATOREN UND AUSDRÜCKE

- Codebeispiel wird angepasst
  - k4 wird kein neu erzeugtes Objekt zugewiesen
  - k4 wird exakt der gleiche Wert zugewiesen, der in k3 gespeichert ist
  - k3 ist eine Referenz auf ein Objekt vom Typ „Kunde“
  - k4 hat nach der Anweisung k4 = k3 dieselbe Referenz als Wert wie k3
- Damit liefert die Operation k3 == k4 das Ergebnis true.

OPERATOREN UND AUSDRÜCKE

k3 wird die Referenz auf ein neues Objekt vom Typ Kunde zugewiesen

k4 wird der exakt gleiche Wert wie k3 zugewiesen (eine Referenz auf ein Objekt vom Typ Kunde)

```
Kunde k3, k4;

boolean e8;

k3 = new Kunde();

k4 = k3;

e8 = k3 == k4;

System.out.println(e8);
```

Deklaration der Variablen k3 und k4, Datentyp ist Kunde, d.h. kein primitiver Datentyp, sondern ein Referenzdatentyp

Der Variablen e8 wird das Ergebnis des Vergleichs k3 == k4 zugewiesen

Wert von e8 wird in der Konsole ausgegeben

OPERATOR ZUR VERBINDUNG VON ZEICHENKETTEN

MEHRERE ZEICHENKETTEN ZUSAMMENFÜHREN

KONKATENATION VON MEHREREN STRINGS MITEINANDER

OPERATOREN UND AUSDRÜCKE

Ausgeführte Operation	Operator	Anwendbar auf die Datentypen	Beispiel
Verketten von Zeichen (Konkatenation) Verketten von mehreren Zeichenketten zu einer neuen Zeichenkette	+	String	String s1, s2 s3 ,s4; s1 = „Hallo“; s2 = „ “; s3 = „Welt“; s4 = s1 + s2 + s3

PRIORITÄTEN

BINDUNGSSTÄRKE BZW. VORRANG DER OPERATOREN

OPERATOREN UND AUSDRÜCKE

- **Priorität:** Bindungsstärke bzw. Vorrang der Operator
  - Auswertungsreihenfolge folgt der Priorität
  - Regelt Vorrang bei unterschiedlichen Operatoren

- Priorität der "Punkt-Operatoren" (*, /, %) höher als Priorität der "Strich-Operatoren" (+, -)
  - d. h. Auswertung so, wie in der Mathematik üblich

- Beispiel: Multiplikation vor Addition, also
  2 + 3 * 4 → 2 + 12 → 14

OPERATOREN UND AUSDRÜCKE

- Klammern (...) erzwingen Auswertungsreihenfolge
  - gewünschte Auswertungsreihenfolge nicht immer gemäß Punkt vor Strich
  - Eingeklammerte Teilausdrücke immer zuerst ausgerechnet
  - Klammern sind um jeden Ausdruck erlaubt
- Beispiele:
  - (2 + 3) * 4 ⟶ 5 * 4 ⟶ 20
  - 2 + (3 * 4) ⟶ 2 + 12 ⟶ 14
  - (2 + 3) ⟶ 5
  - (((2))) ⟶ 2

OPERATOREN UND AUSDRÜCKE

- **Assoziativität** regelt Vorrang bei gleichrangigen Operatoren

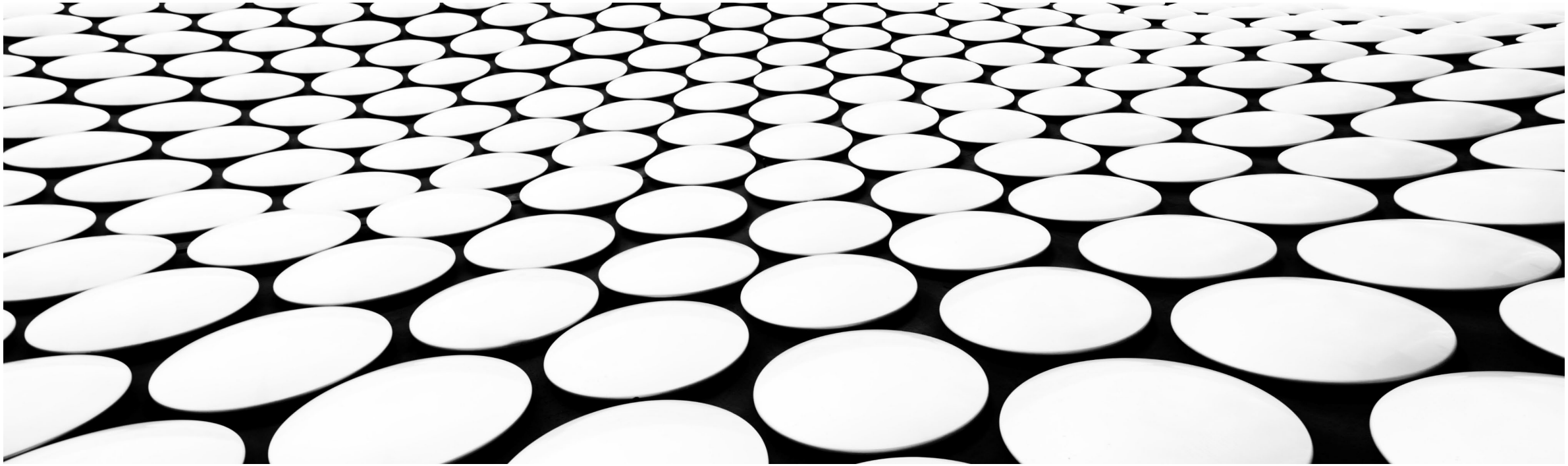
- **Beispiel:** 8-3-2
  - Linker Subtraktionsoperator zuerst: 8 - 3 - 2 → (8 - 3) - 2 → 5 - 2 → 3
  - Rechter Subtraktionsoperator zuerst: 8 - 3 - 2 → 8 - (3 - 2) → 8 - 1 → 7

OPERATOREN UND AUSDRÜCKE

- Charakteristische Assoziativität eines Operators
  - Links-assoziativ: am weitesten links stehender Operator wird zuerst ausgewertet
  - Rechts-assoziativ analog
- Alle binären arithmetischen Operatoren sind linksassoziativ.
  - Also 8 - 3 - 2 ⟶ (8 - 3) - 2 ⟶ 5 - 2 ⟶ 3

OPERATOREN UND AUSDRÜCKE

Operator	Priorität	Assoziativität	Operanden	Bedeutung
+	1	rechts	1	Positives Vorzeichen
-	1	rechts	1	Negatives Vorzeichen
*	2	links	2	Multiplikation
/	2	links	2	Division
%	2	links	2	Modulo
+	3	links	2	Addition
-	3	links	2	Subtraktion



PRIMITIVE DATENTYPEN

Zahlen, Ganze Zahlen, Wahr/Falsch, Fließkomma, Zeichen, Zeichenketten

PRIMITIVE DATENTYPEN

- Der Datentyp eines Attributes bestimmt, welche Informationen in einem Attribut abgelegt werden dürfen.
- Wird als Datentyp eines Attributes der Name einer Klasse angegeben, dürfen nur Objekte dieser Klasse als Werte diesem Attribut zugewiesen werden
- **Primitiven Datentypen** sind sehr einfache Datentypen, die nicht durch eine eigene Klasse beschrieben werden

PRIMITIVE DATENTYPEN

- Primitive Datentypen speichern
  - ganze Zahlen (1, 12, 13131)
  - Fließkommazahlen (1.123, 21234.1232)
  - Wahrheitswerte (true, false)
  - einzelne Zeichen (t, w, f, d)
- Primitive Datentypen sind einfache Standard-Datentypen, die es auch in anderen Programmiersprachen gibt

GANZE ZAHLEN

GANZE ZAHLEN

Schlüsselwort	Größe	Wertebereich
byte	8 Bit (1 Byte)	-128 bis 127 -2^7 bis 2^7-1
short	16 Bit (2 Byte)	-32.768 bis 32.767 -2^{15} bis $2^{15}-1$
int	32 Bit (4 Byte)	-2.147.483.648 bis 2.147.483.647 -2^{31} bis $2^{31}-1$
long	64 Bit (8 Byte)	-9.223.372.036.854.775.808 bis 9.223.372.036.854.775.807 -2^{63} bis $2^{63}-1$

GANZE ZAHLEN - BYTE

```java
byte byte1 = 10;
System.out.println("byte1: " + byte1);
// byte1: 10

byte byteMin = Byte.MIN_VALUE;
System.out.println("byteMin: " + byteMin);
// byteMin: -128

byte byteMax = Byte.MAX_VALUE;
System.out.println("byteMax: " + byteMax);
// byteMax: 127
```

GANZE ZAHLEN - SHORT

```java
short short1 = 10;
System.out.println("short1: " + short1);
// short1: 10

short shortMin = Short.MIN_VALUE;
System.out.println("shortMin: " + shortMin);
// shortMin: -32768

short shortMax = Short.MAX_VALUE;
System.out.println("shortMax: " + shortMax);
// shortMax: 32767
```

GANZE ZAHLEN - INT

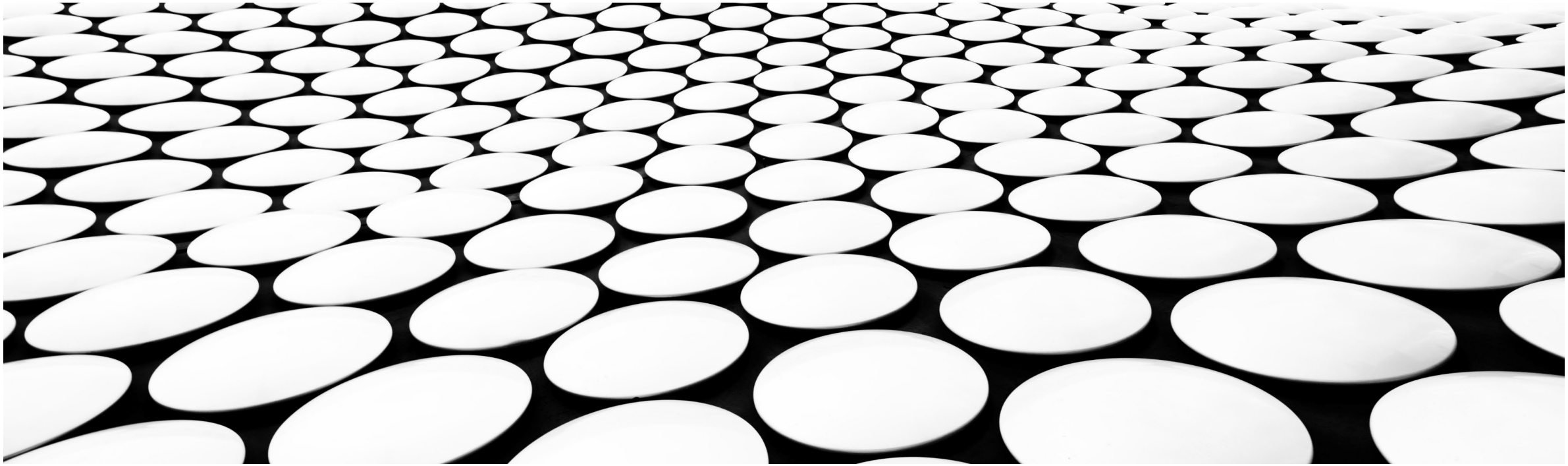
```java
int int1 = 30;
System.out.println("int1: " + int1);
// int1: 30

int intMin = Integer.MIN_VALUE;
System.out.println("intMin: " + intMin);
// intMin: -2147483648

int intMax = Integer.MAX_VALUE;
System.out.println("intMax: " + intMax);
// intMin: 2147483647
```

GANZE ZAHLEN - LONG

```java
long long1 = 10;
System.out.println("long1: " + long1);
// long1: 10

long long2 = 10000000000; //Kompilierfehler
// The literal 10000000000 of type int is out of range

long long3 = 10000000000l;
System.out.println("long3: " + long3);
// long3: 10000000000

long long4 = 10000000000L;
System.out.println("long4: " + long4);
// long4: 10000000000
```

GANZE ZAHLEN - LONG

```java
long longMin = Long.MIN_VALUE;
System.out.println("longMin: " + longMin);
// longMin: -9223372036854775808

long longMax = Long.MAX_VALUE;
System.out.println("longMax: " + longMax);
// longMax: 9223372036854775807
```



KONTROLLSTRUKTUREN

Um der Herr der Ringe – Äh Alligator… Aphoris… Ach was soll´s um den Kram zu Steuern.

KONTROLLSTRUKTUREN

- Reihenfolge der Anweisungen innerhalb eines Methodenrumpfes ist vorgegeben durch
  - Reihenfolge der implementierten Anweisungen

- Kontrolliertes wiederholtes Ausführen von Anweisungen durch Kontrollstrukturen möglich
- Die wichtigsten Kontrollstrukturen sind:
  - bedingte Verzweigungen (if-else) und
  - Schleifen (for, while, do-while)

BEDINGTE VERZWEIGUNG

BEDINGTE VERZWEIGUNG

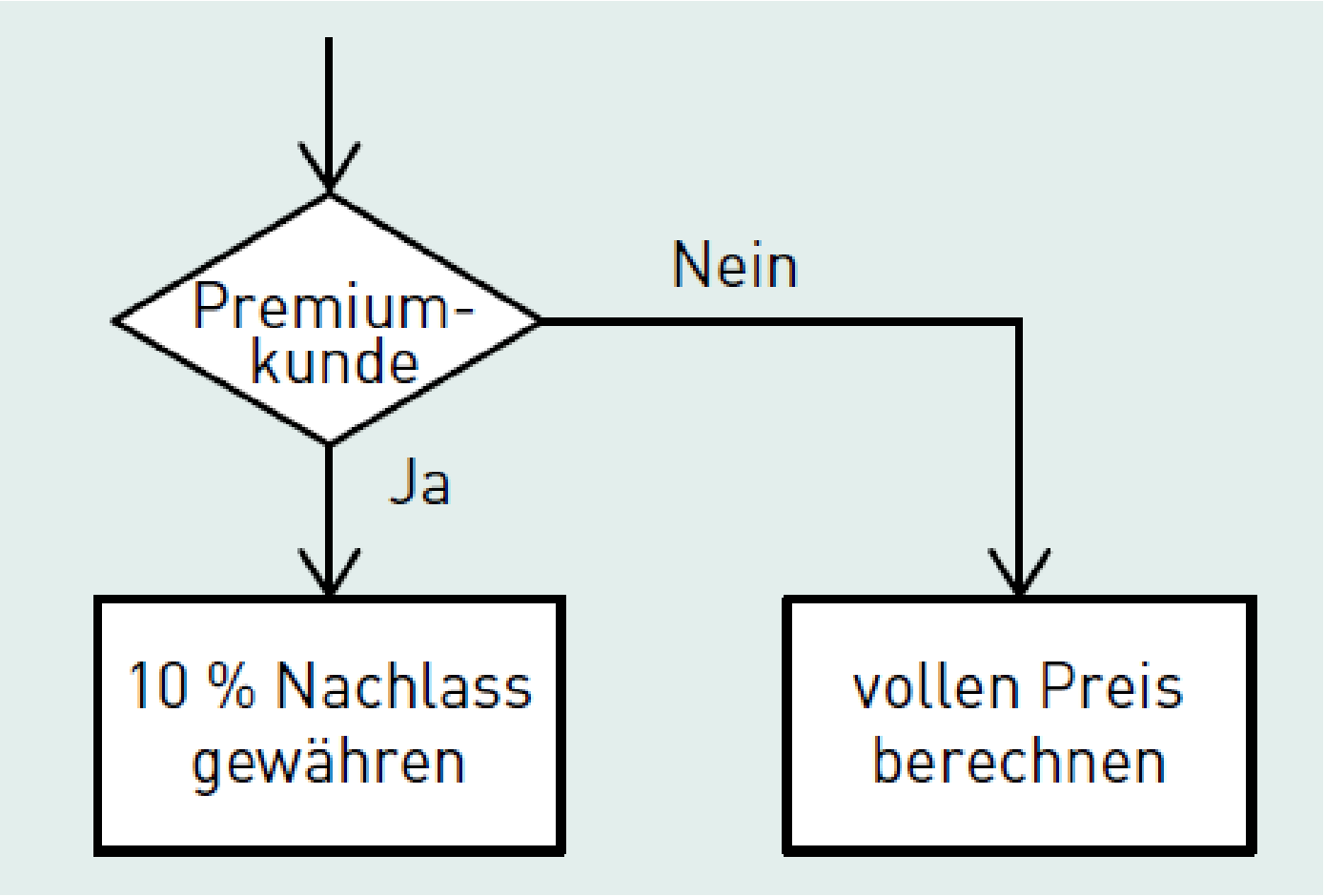
Bedingte Verzweigung wird verwendet um die konkrete Stelle mit der das Programm fortgesetzt wird anhand einer Bedingung zu prüfen.

Struktur:

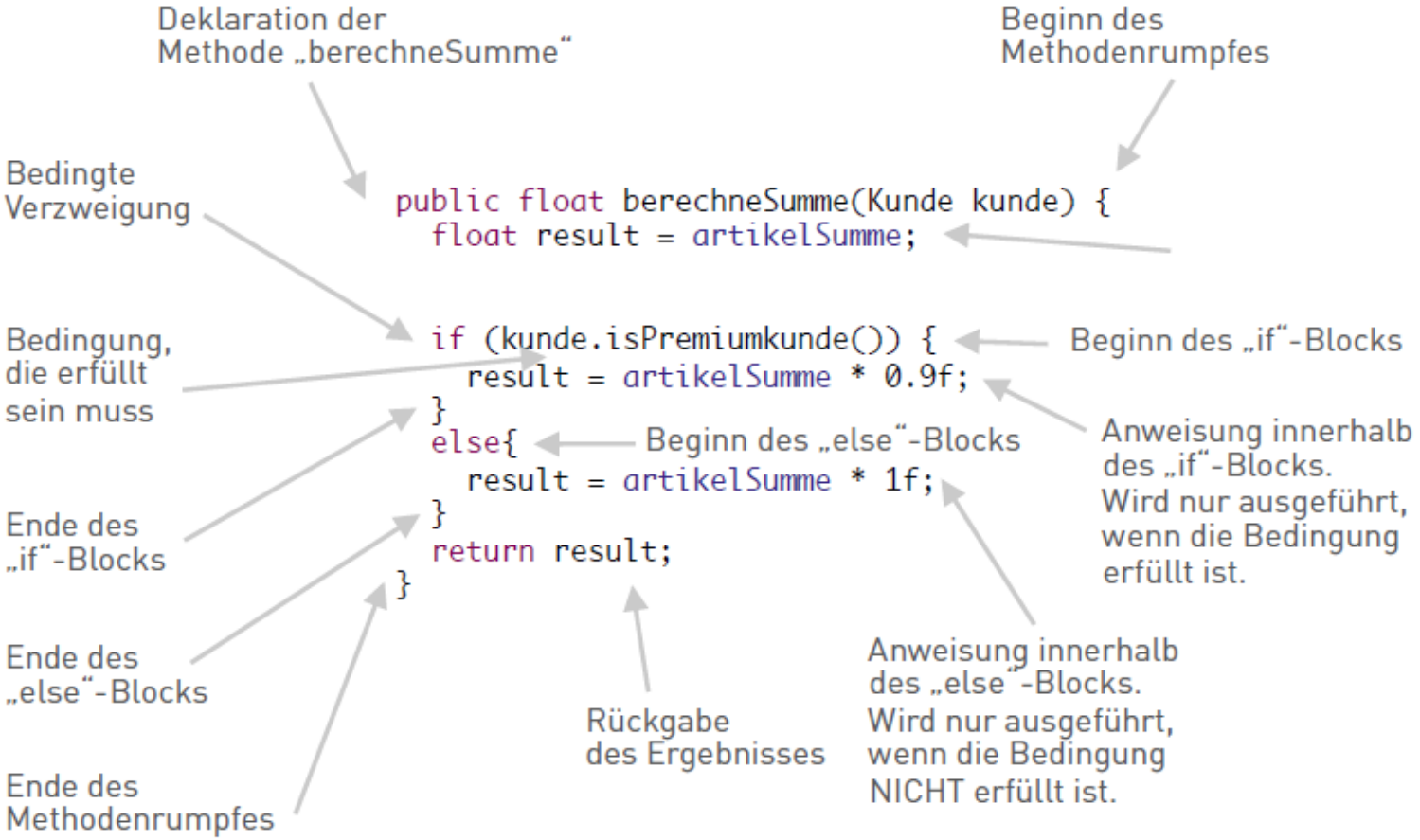
```
if   (Bedingung) {
     Anweisung1;
}
else {
     Anweisung2;
}
```

BEDINGTE VERZWEIGUNG

Anwendungsfall Darstellung im PAP:



BEDINGTE VERZWEIGUNG



ERWEITERTE IF-ELSE VERZWEIGUNG

ERWEITERTE IF-ELSE VERZWEIGUNG

Erweiterten if-else Verzweigung

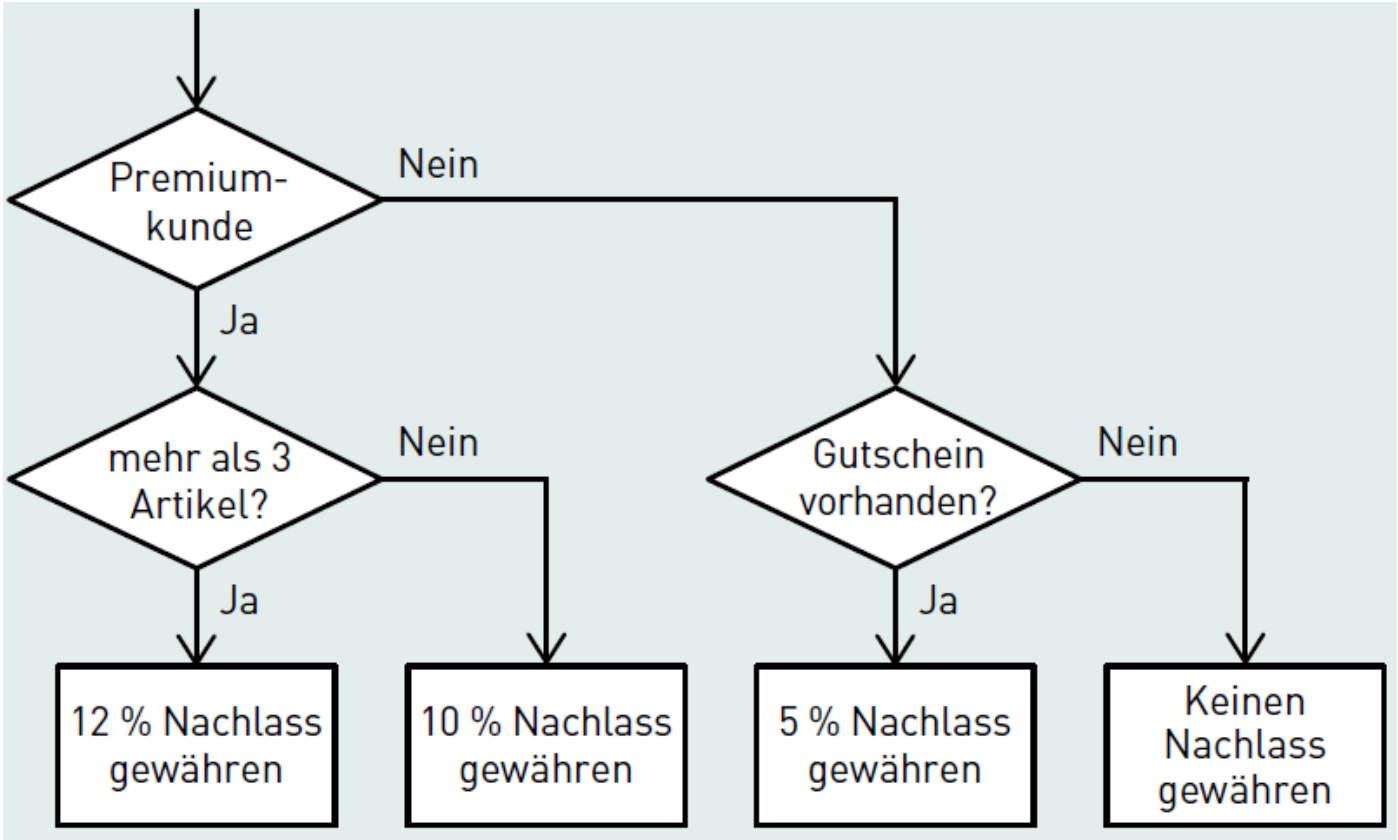
- nicht nur eine Bedingung vor dem else-Block
- mehrere sich ausschließende Bedingungen

Struktur:

```
if (Bedingung1) {
    Anweisung1;
}
else if (Bedingung2) {
    Anweisung2;
}
else {
   Anweisung3;
}
```

ERWEITERTE IF-ELSE VERZWEIGUNG

Anwendungsfall Darstellung im PAP:



ERWEITERTE IF-ELSE VERZWEIGUNG

```
private int anzahlArtikel;
private float artikelSumme;
private boolean gutscheinEingeloest;

public float berechneSumme(Kunde kunde) {

        float result = artikelSumme;

        if (kunde.isPremiumkunde()) {
                if (anzahlArtikel > 3) {
                        result = artikelSumme * 0.88f;
                }
                else {
                        result = artikelSumme * 0.90f;
                }
        }
        else {
                if (gutscheinEingeloest) {
                        result = artikelSumme * 0.95f;
                }
        }

        return result;
}
```

SCHLEIFEN

WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN.....

SCHLEIFEN

- Schleifen sind eine weitere wichtige Kontrollstruktur
  - ermöglichen die mehrfache Ausführung von gleichen Anweisungen hintereinander
  - Anzahl der Schleifendurchläufe bestimmt von einer Schleifenbedingung (auch: Laufbedingung, Abbruchbedingung)

- Drei verschiedene Schleifenarten:
  - While-Schleife,
  - Do-while-Schleife und
  - For-Schleife

WHILE

WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN.....

WHILE

- **while-Schleife** prüft zuerst die Schleifenbedingung.
  - Bedingung erfüllt: Anweisungen werden ausgeführt
  - Andernfalls werden die Anweisungen übersprungen

- Nach Ausführung der Anweisungen erneute Überpüfung der Bedingung
  - Auswertung zu false: Schleife wird beendet
  - Auswertung zu true: Anweisungen der Schleife erneut durchlaufen.

WHILE

Die Struktur einer **while-Schleife**:

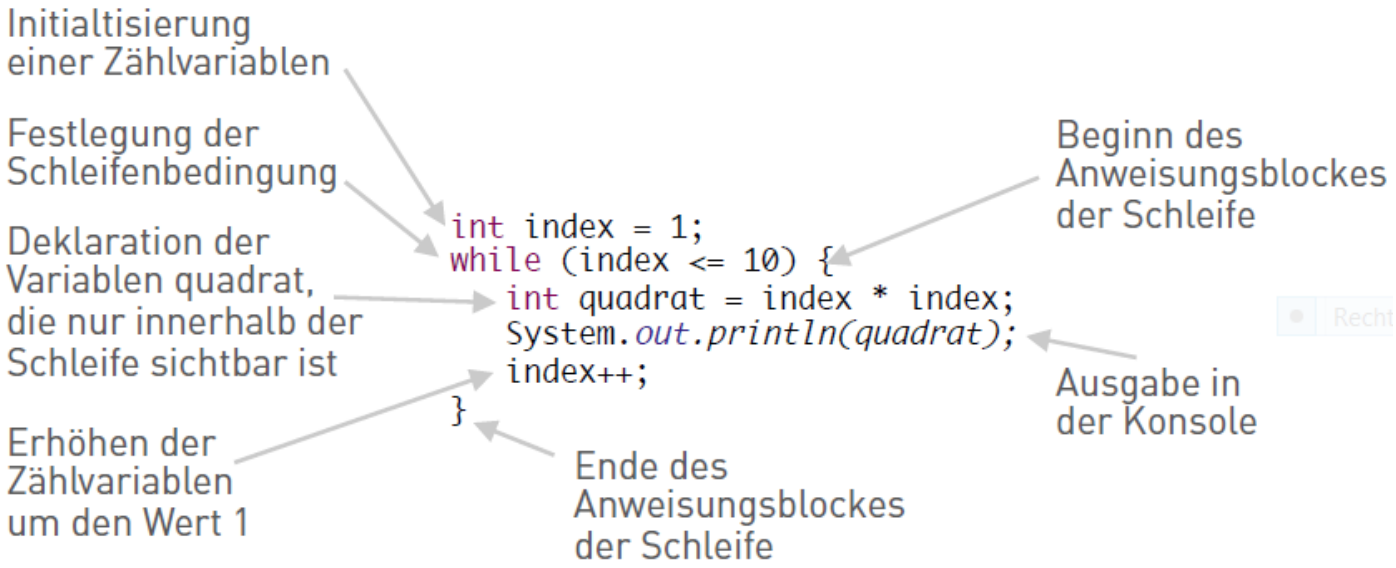
```
while (Bedingung) {
  Anweisungen;
}
```

while-Schleife auch kopfgesteuerte Schleife genannt

- bereits vor dem ersten Ablauf wird die Bedingung geprüft
- Auswertung zu false: komplette Schleife wird übersprungen

WHILE

Beispiel einer while -Schleife



DO-WHILE

WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN.....

DO-WHILE

do-while-Schleife ist eine fußgesteuerte Schleife

- Anweisungen werden mindestens einmal ausgeführt
- Erst dann wird die Schleifenbedingung geprüft
  - Bedingung erfüllt: Anweisungen werden wiederholt
  - Bedingung nicht erfüllt: Schleife wird beendet

Struktur:

```
do {
    Anweisungen;
} while (Bedingung)
```

DO-WHILE

Beginn des Anweisungsblockes der Schleife

```
int index = 1;
do {
    int quadrat = index * index;
    System.out.println(quadrat);
    index++;
} while (index <= 10);
```

Festlegung der Schleifenbedingung

FOR - SCHLEIFE

WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN, WIEDERHOLUNGEN.....

FOR

Elemente des Schleifenkopfes bei der **for-Schleife**:

- die Initialisierung
- Bedingungsprüfung
- das Ändern der Zählvariablen

for-Schleife ist eine kopfgesteuerte Schleife

- vor dem erstmaligen Ausführen wird Bedingung geprüft

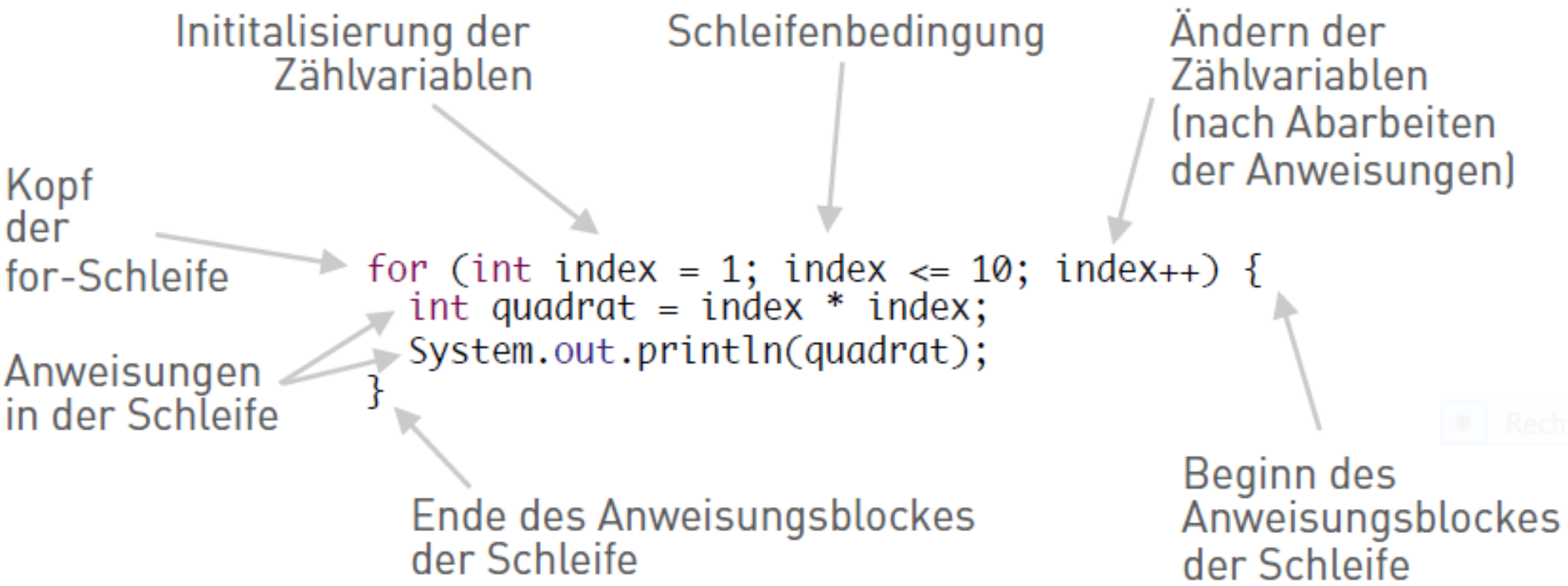
Struktur:

```
for (Initialisierung; Bedingung; Schleifenfortschaltung) {
  Anweisungen;
}
```

FOR

- Ablaufschema einer for-Schleife:
  1. Ausführen Anweisung der Initialisierung
  2. Prüfung der Bedingung
  3. Auswerten der Bedingung
     - Wenn Bedingung true: Ausführung aller Anweisungen der for- Schleife
     - Wenn Bedingung false: Abbruch und keine Ausführung der Anweisungen
  4. Ausführen der Anweisung der Schleifenfortschaltung
  5. Weiter mit Punkt 2 (Prüfung der Bedingung)

FOR


Inititalisierung der Zählvariablen
Schleifenbedingung
Ändern der Zählvariablen (nach Abarbeiten der Anweisungen)
Kopf der for-Schleife
Anweisungen in der Schleife
for (int index = 1; index <= 10; index++) {
int quadrat = index * index;
System.out.println(quadrat);
}
Ende des Anweisungsblockes der Schleife
Beginn des Anweisungsblockes der Schleife

VERSCHACHTELTE KONTROLLSTRUKTUREN

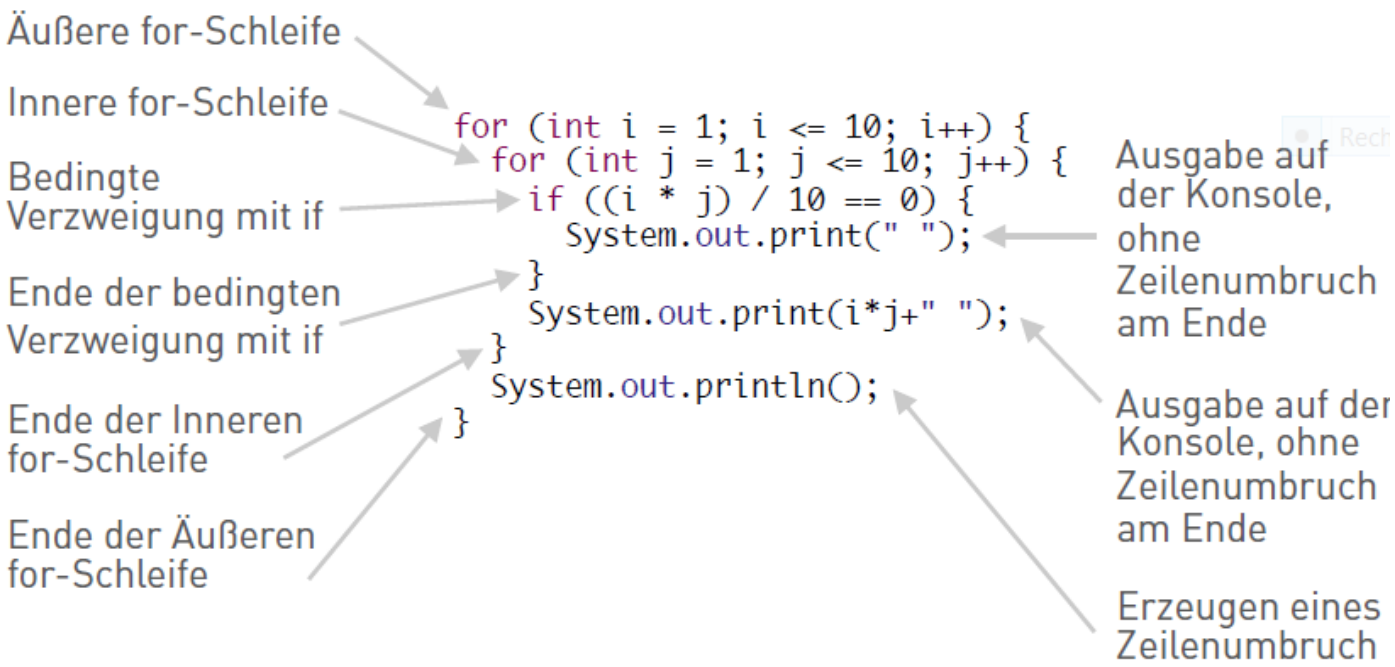
FOR INSIDE AN WHILE INSIDE AN IF INSIDE AN FOR INSIDE AN DO-WHILE... INSANE.

VERSCHACHTELTE FOR - SCHLEIFEN

- Kontrollstrukturen können Kontrollstrukturen enthalten
- Schleifen können ineinander geschachtelt werden
- Codebeispiel (s. nächste Folie) mit zwei verschachtelten for-Schleifen
  - Variablen aus äußeren Kontrollstrukturen: auch in inneren Kontrollstrukturen verfügbar
  - Auf die Variable i kann auch innerhalb der zweiten for-Schleife zugegriffen werden
  - Anweisungen der ersten for-Schleife können nicht auf die Variable j der inneren for-Schleifen zugreifen

VERSCHACHTELTE FOR - SCHLEIFEN

Beispiel mit zwei verschachtelten for-Schleifen



BREAK UND CONTINUE

AWESOME TRY TO CONTROL THE INSANE.

BREAK UND CONTINUE

- break and continue ist in Java mehrdeutig
  - bei zwei ineinander verschachtelten Schleifen, bricht break nur die innere Schleife ab
    - Wie soll die äußere Schleife abgebrochen werden?
  - bei zwei ineinander verschachtelten Schleifen, setzt continue nur die innere Schleife fort
    - Wie soll die äußere Schleife fortgesetzt1 werden?
  - switch – Anweisung benutzt ebenfalls break
    - Wie soll eine Schleife, die eine switch Anweisung enthält, abgebrochen werden?

BREAK

```
    5     1     5
    5     2    10
    5     3    15
    5     4    20
    5     5    25
BREAK
    6     0     0
    6     1     6
    6     2    12
    6     3    18
```

```java
public static void demoBreak() {
    int product = 0;
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            product = i * j;
            System.out.printf("%4d %4d %6d %n", i, j, product);
            if (product == 25) {
                System.out.println("BREAK");
                break;
            }
```

BREAK

```
public static void demoSwitchBreak(){
    for (int i = 1; i < 10; i++) {
        switch (i) {
        case 1:
            System.out.printf("i:%2d %n", i); break;
        case 2:
            System.out.printf("i:%2d    i²: %2d %n", i, i*i); break;
        case 3:
            System.out.printf("i:%2d    i³: %2d %n", i,  i*i*i); break;
        default: System.out.printf("i:%2d ist groesser als drei %n", i); break;
        }
```

```
i: 1
i: 2    i²:  4
i: 3    i³: 27
i: 4 ist groesser als drei
i: 5 ist groesser als drei
i: 6 ist groesser als drei
i: 7 ist groesser als drei
i: 8 ist groesser als drei
i: 9 ist groesser als drei
```

CONTINUE

```
    5     0      0
    5     1      5
    5     2     10
    5     3     15
    5     4     20
CONTINUE
    5     6     30
    5     7     35
    5     8     40
    5     9     45
    5    10     50
```

```java
public static void demoContinue(){
    int product = 0;
    for (int i = 0; i < 100; i++) {
        for (int j = 0; j < 100; j++) {
            product = i * j;
            if (product == 25) {
                System.out.println("CONTINUE");
                continue;
            }
            System.out.printf("%4d %4d %6d %n", i, j, product);
        }
```

BREAK UND CONTINUE MIT MARKEN

AN LABEL TO RULE THEM ALL.

BREAK UND CONTINUE MIT MARKEN

- Definition einer **Marke** (engl. **Label**)
- Bezeichner der Marke wird mit Doppelpunkt abgeschlossen vor eine Anweisung gesetzt
- Die break Anweisung wird mit der Marke markiert

BREAK

```
public static void demoBreakLabel() {
    int product = 0;
    outer:
    for (int i = 0; i < 100; i++) {
        inner:
        for (int j = 0; j < 100; j++) {
            product = i * j;
            System.out.printf("%4d %4d %6d %n", i, j, product);
            if (product == 25) {
                System.out.println("BREAK");
                break outer;
            }
        }
    }
    System.out.println("Ende der Methode demoBreakLabel()");
}
```

```
   1   19     19
   1   20     20
   1   21     21
   1   22     22
   1   23     23
   1   24     24
   1   25     25
BREAK
Ende der Methode demoBreakLabel()
```

BREAK

```
public static void demoBreakLabel() {
    int product = 0;
    outer:
    for (int i = 0; i < 100; i++) {
        inner:
        for (int j = 0; j < 100; j++) {
            product = i * j;
            System.out.printf("%4d %4d %6d %n", i, j, product);
            if (product == 25) {
                System.out.println("BREAK");
                break inner;
            }
        }
    }
    System.out.println("Ende der Methode demoBreakLabel()");
}
```

```
   0    0     0
   5    1     5
   5    2    10
   5    3    15
   5    4    20
   5    5    25
BREAK
   6    0     0
   6    1     6
   6    2    12
   6    3    18
   6    4    24
   6    5    30
```

BREAK

```
public static void demoSwitchBreakLabel(){
    forloop:
    for (int i = 1; i < 10; i++) {
        switch (i) {
        case 1:
            System.out.printf("i:%2d %n", i); break;
        case 2:
            System.out.printf("i:%2d    i²: %2d %n", i, i*i); break forloop;
        case 3:
            System.out.printf("i:%2d    i³: %2d %n", i,  i*i*i); break;
        default: System.out.printf("i:%2d ist groesser als drei %n", i); break;
        }
```

```
i: 1
i: 2    i²:  4
```

CONTINUE

```
public static void demoContinueLabel(){
    int product = 0;
    outer:
    for (int i = 0; i < 100; i++) {
        inner:
        for (int j = 0; j < 100; j++) {
            product = i * j;
            if (product == 25) {
                System.out.println("CONTINUE OUTER");
                continue outer;
            }
            System.out.printf("%4d %4d %6d %n", i, j, product);
        }
```

```
   5    0      0
   5    1      5
   5    2     10
   5    3     15
   5    4     20
CONTINUE OUTER
   6    0      0
   6    1      6
   6    2     12
   6    3     18
   6    4     24
   6    5     30
```

CONTINUE

```
public static void demoContinueLabel(){
    int product = 0;
    outer:
    for (int i = 0; i < 100; i++) {
        inner:
        for (int j = 0; j < 100; j++) {
            product = i * j;
            if (product == 25) {
                System.out.println("CONTINUE INNER");
                continue inner;
            }
            System.out.printf("%4d %4d %6d %n", i, j, product);
        }
```

```
   5    0     0
   5    1     5
   5    2    10
   5    3    15
   5    4    20
CONTINUE INNER
   5    6    30
   5    7    35
   5    8    40
   5    9    45
   5   10    50
```