

Programmiertechnik 1

Teil 3: Java Anweisungen

Ausdrücke / Operatoren / Ablaufsteuerung

Java Anweisungen: Übersicht

Ein Programm besteht aus Anweisungen (*Statements*): 

- Variablen-Definitionen

`Typ Name = Ausdruck;` 

- Ausdrücke mit darauf folgendem Semikolon

`Ausdruck;` // Ausdruck muss Zuweisung oder Methodenaufruf sein

`;` // Spezialfall leere Anweisung 

- Anweisungsblöcke in geschweiften Klammern

`{Anweisung Anweisung ...}`

`{ }` // Spezialfall leere Anweisung 

- Anweisungen zur Ablaufsteuerung (Kontrollstrukturen)

Verzweigungen: `if-else` `switch-case-default` `try-catch-finally`

Schleifen: `while` `do-while` `for`

Sprünge: `break` `continue` `return` `throw`

Java Ausdrücke: Eigenschaften

Ein Ausdruck (*Expression*) liefert einen Wert

- **elementare Ausdrücke:**

<i>Literal</i>	1234
<i>Name</i>	xyz
<i>Objekterzeugung</i>	new <i>Typ</i> (...)

- **zusammengesetzte Ausdrücke** mit Operatoren, Operanden und Klammern:

a = (b + ~c++ + d) * (double)e + f[i] 

Die Auswertungs-Reihenfolge eines zusammengesetzten Ausdrucks ist abhängig

- > von den Eigenschaften der Operatoren (Vorrang und Assoziativität)
- > von der Klammerung

Der Datentyp eines zusammengesetzten Ausdrucks ist abhängig

- > von den Operatoren
- > von den Datentypen der Operanden



*der Compiler versucht
gemischte Datentypen anzugleichen*

Java Operatoren: Eigenschaften

Ein Operator berechnet einen Wert aus seinen Operanden

- die **Stelligkeit** legt die Anzahl der Operanden fest:

unäre Operatoren haben 1 Operanden *(Postfix- oder Präfix-Notation)*

binäre Operatoren haben 2 Operanden *(Infix-Notation)*

ternäre Operatoren haben 3 Operanden *(Infix-Notation)*

- der **Vorrang** (*Precedence*) legt in Ausdrücken mit mehreren Operatoren die Berechnungsreihenfolge fest, z.B.:

$a + b * c$ bedeutet $a + (b * c)$

Bei gleichem Vorrang ist die Reihenfolge von innen nach außen bzw. nach Assoziativität.

- die **Assoziativität** legt bei mehreren Operatoren gleichen Vorrangs die Berechnungsreihenfolge fest

Zuweisungsoperatoren sind rechts-assoziativ, z.B.:

$a = b = c$ bedeutet $a = (b = c)$



Die anderen binären Operatoren sind links-assoziativ, z.B.:

$a + b + c$ bedeutet $(a + b) + c$

Java Operatoren: Übersicht (1)



- Operatoren mit einem Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang ¹
++	Postfix-Inkrement	unär	– 2	1
--	Postfix-Dekrement	unär	– 2	1
. <i>Komponente</i>	Auswahl	unär	– 2	1
[<i>Index</i>]	Indizierung	unär ³	– 2	1
(<i>Parameterliste</i>)	Methodenaufruf	unär ³	– 2	1
++	Präfix-Inkrement	unär	– 2	2
--	Präfix-Dekrement	unär	– 2	2
+	Unäres Plus	unär	– 2	2
-	Unäres Minus	unär	– 2	2
!	Logische Negation	unär	– 2	2
~	Bitweise Invertierung	unär	– 2	2
(<i>Typ</i>)	Typanpassung	unär	– 2	2

¹ Sortierung vom höchsten Vorrang 1 bis niedrigstem Vorrang 14.

² Einstellige Operatoren haben keine Assoziativität. Sie werden von innen nach außen berechnet.

³ In () oder [] geklammerte Parameter der Operatoren bleiben bei der Stelligkeit unberücksichtigt.

Java Operatoren: Übersicht (2)

- Operatoren mit zwei Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang
*	Multiplikation	binär	links	3
/	Division	binär	links	3
%	Modulo	binär	links	3
+	Addition / Konkatenation	Binär	links	4
-	Subtraktion	binär	links	4
<<	Links-Shift	binär	links	5
>> (bzw. >>>)	Rechts-Shift (mit bzw. ohne Vorz.)	binär	links	5
<	kleiner	binär	links	6
<=	kleiner-gleich	binär	links	6
>	größer	binär	links	6
>=	größer-gleich	binär	links	6
==	Gleichheit	binär	links	7
!=	Ungleichheit	Binär	links	7
&	bitweises (bzw. logisches) Und	binär	links	8
^	bitweises (bzw. logisches) XOR	binär	links	9
	bitweises (bzw. logisches) Oder	binär	links	10

Java Operatoren: Übersicht (3)

- weitere Operatoren mit zwei Operanden, bzw. in einem Fall mit drei Operanden:

Operator	Name	Stelligkeit	Assoziativität	Vorrang
&&	Logisches Und (lazy eval.)	binär	links	11
	Logisches Oder (lazy eval.)	binär	links	12
? :	Bedingung	ternär	rechts	13
=	Zuweisung	binär	rechts ⁴	14
+=	Additions-Zuweisung	binär	rechts ⁴	14
-=	Subtraktions-Zuweisung	binär	rechts ⁴	14
*=	Multiplikations-Zuweisung	binär	rechts ⁴	14
/=	Divisions-Zuweisung	binär	rechts ⁴	14
%=	Modulo-Zuweisung	binär	rechts ⁴	14
^=	Bitweise-XOR-Zuweisung	binär	rechts ⁴	14
=	Bitweise-Oder-Zuweisung	binär	rechts ⁴	14
&=	Bitweise-Und-Zuweisung	binär	rechts ⁴	14
<<=	Links-Shift-Zuweisung	binär	rechts ⁴	14
>>= (bzw. >>>=)	Rechts-Shift-Zuweisung	binär	rechts ⁴	14

⁴ Klammerung entgegen der Assoziativität ist Syntaxfehler.

Java Operatoren: Zuweisungen

Ein Zuweisungsoperator schreibt einen Wert in einen Speicherbereich.

- der einfache Zuweisungsoperator `=` ersetzt den bisherigen Wert:

`Variable = Wert`

`Feld [Index] = Wert`

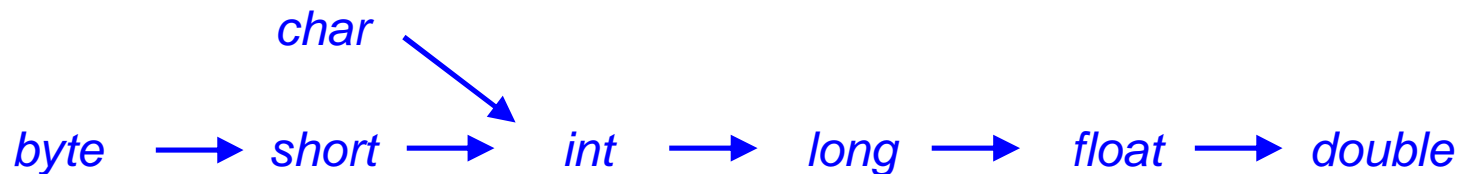
- die zusammengesetzten Zuweisungsoperatoren ändern den bisherigen Wert:

`Variable += Wert`

Kurzschreibweise für: `Variable = Variable + Wert`

(analog `-=`, `=`, `/=`, `%=`, `^=`, `|=`, `&=`, `<<=`, `>>=`, `>>>=`)*

- der **Datentyp** eines Zuweisungsausdrucks ist der Datentyp der linken Seite
*Der Compiler meldet einen Fehler, wenn der Typ der rechten Seite nicht dazu passt!
Evtl. nimmt der Compiler implizite Typanpassung vor z.B. bei Zahltypen:*



- der **Wert** eines Zuweisungsausdrucks ist der neue Wert der linken Seite

Java Operatoren: Inkrement und Dekrement

Ein Inkrement- oder Dekrementoperator ändert einen Zahlwert im Speicher um 1.

- Präfix-Inkrement `++` und -Dekrement `--` sind Kurzschreibweisen, z.B:

`++ Variable` // gleichbedeutend mit `Variable += 1`

`-- Variable` // gleichbedeutend mit `Variable -= 1`

Der **Datentyp** des Ausdrucks insgesamt ist der Datentyp der Variablen.

Der **Wert** des Ausdrucks insgesamt ist der neue Wert der Variablen.



- Postfix-Inkrement und -Dekrement unterscheiden sich nur beim Wert des Ausdrucks von den Präfix-Entsprechungen:

`Variable++`

`Variable--`

Der **Wert** des Ausdrucks insgesamt ist der alte (!) Wert der Variablen.

Java Operatoren: Arithmetik (1)

Arithmetische Operatoren rechnen mit Zahlen.

- unäre Vorzeichenoperatoren für Zahlen **+**, **-**
- binäre Rechenoperatoren für Zahlen **+**, **-**, *****, **/**, **%**

*Der Modulooperator % liefert bei ganzzahliger Division den Rest: $(a/b) * b + a \% b$ ist a*

- unärer Bit-Operator für ganze Zahlen

~ bitweises Komplement



- binäre Bit-Operatoren für ganze Zahlen

& bitweises Und

| bitweises Oder

^ bitweises exklusives Oder

<< Links-Shift (*dabei von rechts mit Nullen auffüllen*)

>> Rechts-Shift (*dabei von links mit dem Vorzeichen auffüllen*)

>>> Logischer Rechts-Shift (*dabei von links mit Nullen auffüllen*)

Java Operatoren: Arithmetik (2)

- der **Datentyp** eines arithmetischen Ausdrucks ist der größte Datentyp seiner Operanden, mindestens aber `int`

größter Datentyp ist der Datentyp mit dem größten Zahlenbereich

- der **Wert** eines arithmetischen Ausdrucks ist das Ergebnis der Berechnung

`10 + 12` // Wert ist 22

`10 | 12` // Wert ist 14

`10 << 2` // Wert ist 40

`-10 >> 2` // Wert ist -3





Logische Operatoren verknüpfen die Wahrheitswerte `true` und `false`.

- unärer logischer Operator

! logisches Nicht

- binäre logische Operatoren

&& logisches Und (*rechten Operand nur berechnen, wenn linker `true`*)

|| logisches Oder (*rechten Operand nur berechnen, wenn linker `false`*)

^ exklusives logisches Oder (*mit vollständiger Auswertung beider Operanden*)

& logisches Und (*mit vollständiger Auswertung beider Operanden*)

| inklusives logisches Oder (*mit vollständiger Auswertung beider Operanden*)

Achtung: `&`, `^` und `|` haben einen höheren Vorrang als `&&` und `||`





- der **Datentyp** eines logischen Ausdrucks ist `boolean`
- der **Wert** eines logischen Ausdrucks ist die Verknüpfung der Operanden

Logische Operatoren werden zum Verknüpfen von Vergleichsausdrücken verwendet:

`a > 0 && a < 5` // *true für a = 1, 2, 3, 4*

Java Operatoren: Vergleiche

Ein Vergleichsoperator prüft eine Relation zwischen zwei Werten.

- Gleichheit / Ungleichheit: `==` `!=`  
- kleiner / größer: `<` `<=` `>` `>=`  
- der **Datentyp** eines Vergleichs-Ausdrucks ist `boolean`
Der Compiler meldet einen Fehler, wenn die Typen der zu vergleichenden Werte nicht zusammenpassen!
- der **Wert** eines Vergleichs-Ausdrucks ist `true`, wenn die Relation zutrifft, sonst `false`
- Vorsicht Falle:

die Gleichheits-Relation wird mit zwei Gleichheitszeichen geschrieben

Variable `==` 5 */* prüft, ob Variable den Wert 5 hat */*  

Variable `=` 5 */* setzt Variable auf den Wert 5 */*

Java Operatoren: Bedingung



Der dreistellige Bedingungsoperator bildet eine Ausdrucks-Alternative.

- Syntax: `Ausdruck1 ? Ausdruck2 : Ausdruck3`
- der **Datentyp** des Ausdrucks insgesamt ist der gemeinsame Datentyp (nach Typanpassung) von *Ausdruck2* und *Ausdruck3*
- der **Wert** des Ausdrucks insgesamt ist der Wert von *Ausdruck2*, falls der Wert von *Ausdruck1* `true` ist
der Wert von *Ausdruck3*, falls der Wert von *Ausdruck1* `false` ist
- der Bedingungsoperator ist rechts-assoziativ:

`a ? b : c ? d : e` bedeutet `a ? b : (c ? d : e)`

Stilempfehlung: Bedingungsoperator höchstens in sehr einfachen Fällen und ohne Verschachtelung verwenden, generell `if-else`-Anweisungen bevorzugen

Java Operatoren: Explizite Typanpassung

Der **Typanpassungsoperator** (*Cast-Operator*) erzwingt den Datentyp eines Ausdrucks.

- Syntax: **(Zieltyp) Ausdruck**
- der **Datentyp** des Ausdrucks insgesamt ist angegebene Zieltyp
Der Compiler meldet einen Fehler, wenn die Anpassung an den Zieltyp unmöglich ist.
- der **Wert** des Ausdrucks insgesamt ist der Wert nach Typanpassung

```
(double) 1      // Wert ist Gleitkommazahl 1.0  
(int) 1.2      // Wert ist ganze Zahl 1  
(int) "Hallo"  // Compiler meldet Fehler
```

Bestimmte Typanpassungen werden erst zur Laufzeit auf Machbarkeit geprüft und können dann zu einer Ausnahme führen.

Java Ausdrücke: Auswertungs-Reihenfolge

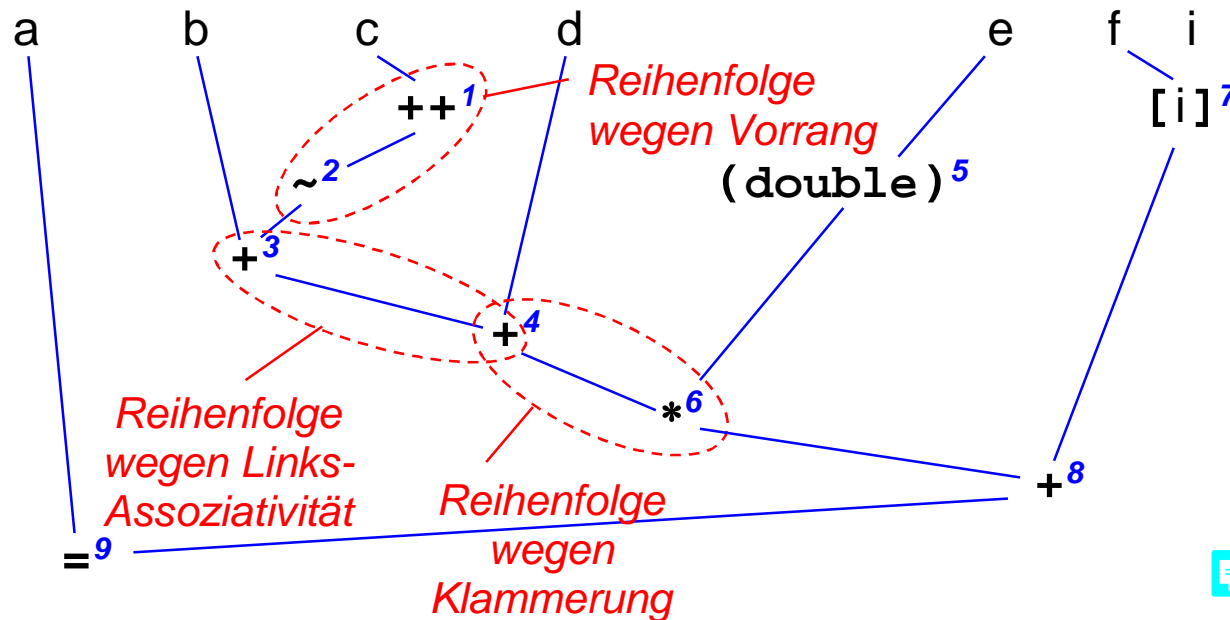
- Die **Auswertungs-Reihenfolge** eines Ausdrucks wird bestimmt von Vorrang, Assoziativität und Klammerung:

a = (b + ~ c ++ + d) * (double) e + f [i]

↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓

14 (4 2 1) 4) 3 2 4 1

- eindeutig darstellbar als **Auswertungsbaum**:



Reihenfolge in jedem Ast von oben nach unten

Reihenfolge zwischen den Ästen von links nach rechts

erst linker Operand, dann rechter Operand ergibt Reihenfolge 1 bis 9

Java Ablaufsteuerung: Verzweigung

Eine Verzweigung ermöglicht optionale und alternative Anweisungen.



- Syntax:

```
if (Bedingung) Anweisung // falls Bedingung erfüllt
```

```
if (Bedingung)  
    Anweisung1 // falls Bedingung erfüllt  
else  
    Anweisung2 // falls Bedingung nicht erfüllt
```

```
if (Bedingung1)  
    Anweisung1 // falls Bedingung1 erfüllt  
else if (Bedingung2)  
    Anweisung2 // falls nur Bedingung2 erfüllt  
else  
    Anweisung3 // falls keine Bedingung erfüllt
```

Eine *Bedingung* ist ein Ausdruck mit Datentyp **boolean**.

Vorsicht bei  
geschachtelten
Verzweigungen:

Ein `else`-Teil gehört immer zum letzten noch offenen `if`.

Eine andere Zuordnung muss mit geschweiften Klammern erzwungen werden:

```
if (Bedingung1) {  
    if (Bedingung2) ...  
} else {  
    ...  
}
```

Beispielprogramm Verzweigung



```
import java.util.Scanner;

public final class Verzweigung {
    private Verzweigung() { }
    private static final Scanner EINGABE = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.print("Zwei Zahlen eingeben: ");
        int m = EINGABE.nextInt();
        int n = EINGABE.nextInt();

        if (m == n) {
            System.out.println("Beide Zahlen sind gleich!");
        } else if (m > n) {
            System.out.printf("Maximum: %d\n", m);
        } else {
            System.out.printf("Maximum: %d\n", n);
        }
    }
}
```

Liest zwei ganze Zahlen ein und gibt deren Maximum aus.

Java Ablaufsteuerung: Fallunterscheidung

Die Fallunterscheidung ist eine spezielle Schreibweise für Mehrfachverzweigungen.

• Syntax:

```
switch (Ausdruck) {  
    case Wert1:  
        Anweisung1  
        break;  
    case Wert2:  
        Anweisung2  
        break;  
    default:  
        Anweisung3  
}
```



Im Prinzip gleichbedeutend mit:

```
if (Ausdruck == Wert1)  
    Anweisung1  
else if (Ausdruck == Wert2)  
    Anweisung2  
else  
    Anweisung3
```



Der *Ausdruck* muss einen ganzzahligen Typ, einen **enum**-Typ oder den Typ String haben. 

Die **case**-Werte müssen dazu passende eindeutige Compilezeit-Konstanten sein. 

Der **default**-Fall wird ausgeführt, wenn der Ausdruck keinen der **case**-Werte hat.

Mit **break** wird die Fallunterscheidung verlassen. *Ohne break z.B. hinter Anweisung1 würde nach Anweisung1 die Anweisung2 ausgeführt*

Beispielprogramm Fallunterscheidung (1)



```
import java.util.Scanner;

public final class Fallunterscheidung {
    private Fallunterscheidung() { }
    private static final Scanner EINGABE = new Scanner(System.in);

    public static void main(String[] args) {
        System.out.print("Monat eingeben [1-12]: ");
        int month = EINGABE.hasNextInt() ? EINGABE.nextInt() : 0;

        switch (month) {
            case 2:
                System.out.println("28 oder 29 Tage");
                break;

            case 4:
            case 6:
            case 9:
            case 11:
                System.out.println("30 Tage");
                break;
```

Gibt die Anzahl der Tage eines Monats aus.

Beispielprogramm Fallunterscheidung (2)

// Fortsetzung ...


```
case 1:  
case 3:  
case 5:  
case 7:  
case 8:  
case 10:  
case 12:  
    System.out.println("31 Tage");  
    break;  
  
default:  
    System.err.println("Eingabefehler!");  
}  
}  
}
```

Beispielprogramm Fallunterscheidung für enum-Werte

```
public final class FallunterscheidungEnum {  
    private FallunterscheidungEnum() { }  
    private enum Month {JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC}  
    public static void main(String[] args) {  
        Month m = Month.valueOf(new java.util.Scanner(System.in).next());  
        switch (m) {  
            case FEB:      System.out.println("28 oder 29 Tage");  
                          break;  
            case APR:  
            case JUN:  
            case SEP:  
            case NOV:     System.out.println("30 Tage");  
                          break;  
            default:      System.out.println("31 Tage");  
        }  
    }  
}
```

Gibt die Anzahl der Tage eines Monats aus.

Beispielprogramm Fallunterscheidung für Strings (1)

```
import java.util.Scanner;   
  
public final class FallunterscheidungString {  
    private FallunterscheidungString() { }  
  
    public static void main(String[] args) {  
        System.out.print("Monat eingeben: ");  
        String month = new Scanner(System.in).next();  
  
        switch (month) {  
            case "Februar":  
                System.out.println("28 oder 29 Tage");  
                break;  
  
            case "April":  
            case "Juni":  
            case "September":  
            case "November":  
                System.out.println("30 Tage");  
                break;
```

*Gibt die Anzahl der Tage
eines Monats aus.*

Beispielprogramm Fallunterscheidung für Strings (2)

// Fortsetzung ...

```
case "Januar":
case "Maerz":
case "Mai":
case "Juli":
case "August":
case "Oktober":
case "Dezember":
    System.out.println("31 Tage");
    break;

default:
    System.err.println("Eingabefehler!");
}
}
}
```


Java Ablaufsteuerung: Schleifen (1)

Eine Schleife ermöglicht die wiederholte Ausführung einer Anweisung.

- Syntax der **while**-Schleife:

```
while (Bedingung)  
    Anweisung
```

Wiederholt die Anweisung, solange die Bedingung gilt.

- Syntax der **do**-Schleife:

```
do  
    Anweisung  
while (Bedingung);
```



Führt die Anweisung aus und wiederholt sie dann, solange die Bedingung gilt.

Gleichbedeutend mit:

```
{  
    Anweisung  
    while (Bedingung)  
        Anweisung  
}
```

Eine *Bedingung* ist wie gehabt ein Ausdruck mit Datentyp `boolean`.

Java Ablaufsteuerung: Schleifen (2)

Die **for**-Schleife ist eine spezielle Schreibweise für Schleifen mit **Laufvariablen**.
Schleifen werden häufig benutzt, um Felder oder Listen (allgemein: Aggregate) abzulaufen. Dabei werden die aggregierten Elemente über eine Laufvariable angesprochen.

- Syntax der allgemeinen **for**-Schleife:

```
for (Initialisierung; Bedingung; Fortschaltung)  
    Anweisung
```


Die *Initialisierung* ist ein Ausdruck, der die Laufvariable auf das erste Element des Aggregats setzt.

Die *Fortschaltung* ist ein Ausdruck, der die Laufvariable auf das nächst folgende Element des Aggregats setzt.

Die *Bedingung* prüft, ob alle Elemente besucht wurden.

Gleichbedeutend mit:

```
{  
    Initialisierung;  
    while (Bedingung) {  
        Anweisung  
        Fortschaltung;  
    }  
}
```



- Syntax der vereinfachten **for-each**-Schleife:

```
for (Typ Laufvariable : Aggregat)  
    Anweisung
```



Aggregate sind Sammlungen von Werten gleichen Typs, z.B. Felder

Beispielprogramm while-Schleife



```
import java.util.Scanner;

public final class WhileSchleife {
    private WhileSchleife() { }
    private static final Scanner EINGABE = new Scanner(System.in);

    public static void main(String[] args) {
        int sum = 0;
        System.out.println("Ganze Zahlen eingeben (Ende mit Strg-Z):");
        while (EINGABE.hasNextInt()) {
            sum += EINGABE.nextInt();
        }
        System.out.printf("Summe: %d\n", sum);
    }
}
```

Liest ganze Zahlen ein und gibt deren Summe aus.

Beispielprogramm do-Schleife

```
import java.util.Scanner;

public final class DoSchleife {
    private DoSchleife() { }
    private static final Scanner EINGABE = new Scanner(in);

    public static void main(String[] args) {
        int n = 0;
        do {
            System.out.println("Zahl zwischen 0 und 255 eingeben: ");
            n = EINGABE.nextInt();
        } while (n < 0 || n > 255);

        System.out.print(" "); // sieben Leerzeichen
        do {
            System.out.printf("%d\b\b", n % 2);
            n /= 2;
        } while (n > 0);
        System.out.println();
    }
}
```

Liest eine ganze Zahl ein und gibt sie in Binärdarstellung aus.

Beispielprogramm for-Schleife

```
public final class ForSchleife {  
    private ForSchleife() { }  
  
    public static void main(String[] args) {  
        double[] anArray = {3.625, 3.648, 3.853, 4.042};  
  
        for (int i = 0; i < anArray.length; ++i) {  
            System.out.println(anArray[i]);  
        }  
  
        for (double n : anArray) {  
            System.out.println(n);  
        }  
    }  
}
```

Gibt zweimal alle Feldelemente aus.

*i ist Laufvariable
für Feldindices*

n ist Laufvariable für Feldelemente

*Die Laufvariablen i und n sind jeweils nur innerhalb ihrer Schleife bekannt.
Bei i könnte man dies ändern, indem man i vor der Schleife definiert und
im Schleifenkopf nur noch initialisiert.*

Java Ablaufsteuerung: Ausnahmen



Ausnahmebehandlung (*Exception Handling*) ermöglicht es, normalen Ablauf und Ausnahmefälle (das sind in der Regel Fehler) zu trennen.

- Syntax:

```
try {  
    Anweisung1 // normaler Ablauf  
} catch (Ausnahmedeklaration) {  
    Anweisung2 // Fehlerbehandlung  
} finally {  
    Anweisung3  
}
```

*Ausnahmedeklarationen bestehen aus einem Ausnahmetyp und einem Variablennamen
oder mehreren durch | getrennten Ausnahmetypen und einem Variablennamen (Multi-catch)*

Nach einem `try`-Block dürfen mehrere `catch`-Blöcke folgen.

bei einer Ausnahme wird der erste passende `catch`-Blöcke ausgeführt (Suche von oben nach unten)




Der `finally`-Block darf fehlen.

der `finally`-Block wird (sofern vorhanden) unabhängig vom Auftreten einer Ausnahme immer als letztes ausgeführt

Beispielprogramm Ausnahmebehandlung (1)


```
import java.util.Scanner;  
import java.util.InputMismatchException;  
import java.util.NoSuchElementException;
```

Gibt die Anzahl der Tage eines Monats aus.

```
public final class Ausnahmebehandlung {   
    private Ausnahmebehandlung() { }  
    private static final Scanner EINGABE = new Scanner(System.in);  
  
    public static void main(String[] args) {  
        System.out.print("Monat eingeben [1-12]: ");  
  
        try {  
            int month = EINGABE.nextInt();   
            if (month < 1 || month > 12) {  
                throw new Exception("Fehler: kein Monat");   
            }  
  
            switch (month) {  
                case 2:  
                    System.out.println("28 oder 29 Tage");  
                    break;
```

Beispielprogramm Ausnahmebehandlung (2)

```
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("30 Tage");
        break;
    default:
        System.out.println("31 Tage");
    }
} catch (InputMismatchException x) { // Ausnahme in nextInt()
    System.err.println("Fehler: keine Zahl");
} catch (NoSuchElementException x) { // Ausnahme in nextInt()
    System.err.println("Fehler: keine Eingabe");
} catch (Exception x) {
    System.err.println(x.getMessage()); // siehe oben throw
} finally {
    EINGABE.close();
}
}
```




Java Ablaufsteuerung: Sprünge (1)

Eine **break**-Anweisung springt hinter die umgebende Fallunterscheidung / Schleife:

- Syntax des einfachen **break**:

```
while (...) {  
    ...  
    if (Bedingung) break;  
    ...  
}  
... // break springt hier hin
```

- Syntax des **break** mit Label:

```
einLabel:   
for (...) {  
    for (...) {  
        if (Bedingung) break einLabel;  
    }  
}  
... // break springt hier hin
```

Gleichbedeutend mit:

```
boolean stop = false;  
while (... && !stop) {  
    ...  
    if (Bedingung) {  
        stop = true;  
    } else {  
        ...  
    }  
}
```

Die Variante mit Label erlaubt es, ineinander geschachtelte Schleifen (oder switch-case-Anweisungen) in einem Schritt zu verlassen.

Java Ablaufsteuerung: Sprünge (2)

Eine **continue**-Anweisung springt zum nächsten Schleifen-Durchlauf, d.h. bei einer `do`- oder `while`-Schleife zur Auswertung der Fortsetzungs-Bedingung und bei einer `for`-Schleife zur Fortschaltung

- Syntax des einfachen `continue`:

```
while ( /* continue springt hier hin */ ... ) {  
    if ( Bedingung ) continue;  
    ...  
}
```

- Syntax des `continue` mit Label:

einLabel:

```
for ( ...; ...; /* continue springt hier hin */ ... ) {  
    for ( ...; ...; ... ) {  
        if ( Bedingung ) continue einLabel;  
        ...  
    }  
}
```

Die Variante mit Label erlaubt es, bei geschachtelten Schleifen die innere Schleife zu beenden und die markierte äußere fortzusetzen.

Gleichbedeutend mit:

```
while ( ... ) {  
    if ( !Bedingung ) {  
        ...  
    }  
}
```

Java Ablaufsteuerung: Sprünge (3)

Eine **return**-Anweisung springt an die Aufrufstelle einer Methode zurück.
Genauerer später bei den Methoden.

- Innerhalb von `main` beendet `return` das Programm:

```
public static void main(String[] args) {  
    ...  
    if (Bedingung) return;  
    ...  
}
```

Gleichbedeutend mit:

```
public static void main(String[] args) {  
    ...  
    if (!Bedingung) {  
        ...  
    }  
}
```

*Am Ende von `main` fügt
der Compiler automatisch
ein `return` ein.*

Java Ablaufsteuerung: Sprünge (4)

Eine **throw**-Anweisung springt in den nächsten passenden **catch**-Block:

- Syntax:

```
try {  
    throw new Ausnahme();  
} catch (Ausnahme x) {  
    ... // throw springt hier hin  
}
```

*Genaueres zu Ausnahmen
siehe später bei den
Methoden und Klassen*

Wenn kein **catch**-Block zum Typ der Ausnahme passt
oder wenn **throw** nicht in einem **try**-Block steht:

- > wird ein **return** ausgeführt
- > und dann wiederum nach einem passenden **catch**-Block gesucht
- > usw.

Wird nirgendwo ein passender **catch**-Block gefunden,
beendet die **throw**-Anweisung das Programm.

Beispielprogramm Sprünge

```
import java.util.Scanner;
public final class Spruenge {
    private Spruenge() { }
    private static final Scanner EINGABE = new Scanner(in);
    public static void main(String[] args) {
        int sum = 0;
        System.out.println("Ganze Zahlen eingeben (Ende mit Strg-D oder =):");
        while (EINGABE.hasNext()) {
            if (!EINGABE.hasNextInt()) {
                String s = EINGABE.next();
                if (s.equals("=")) break; // hinter die Schleife springen
                System.err.printf("Folgende Eingabe wird ignoriert: %s%n", s);
                continue; // zum nächsten Schleifendurchlauf springen
            }
            sum += EINGABE.nextInt();
        }
        System.out.printf("Summe: %d%n", sum);
        return; // aus dem Programm springen (normales Programmende)
    }
}
```

Liest ganze Zahlen ein und gibt deren Summe aus.

Java Anweisungen: Empfehlungen (1)

- Leerzeichen machen Ausdrücke lesbarer, unnötige Klammern nicht unbedingt:

`a + b * c`

~~`a+(b*c)`~~

// Klammern unnötig

`(a + b) * c`

~~`(a+b)*c`~~

// Klammern notwendig

- Ausdrücke mit Seiteneffekten vermeiden:

~~`a = b + c++;`~~

// Seiteneffekt auf c

`a = b + c;`

// Aufteilung meistens besser

`++c;`

- Nur eine Anweisung pro Zeile schreiben, Kontrollstrukturen mehrzeilig schreiben.

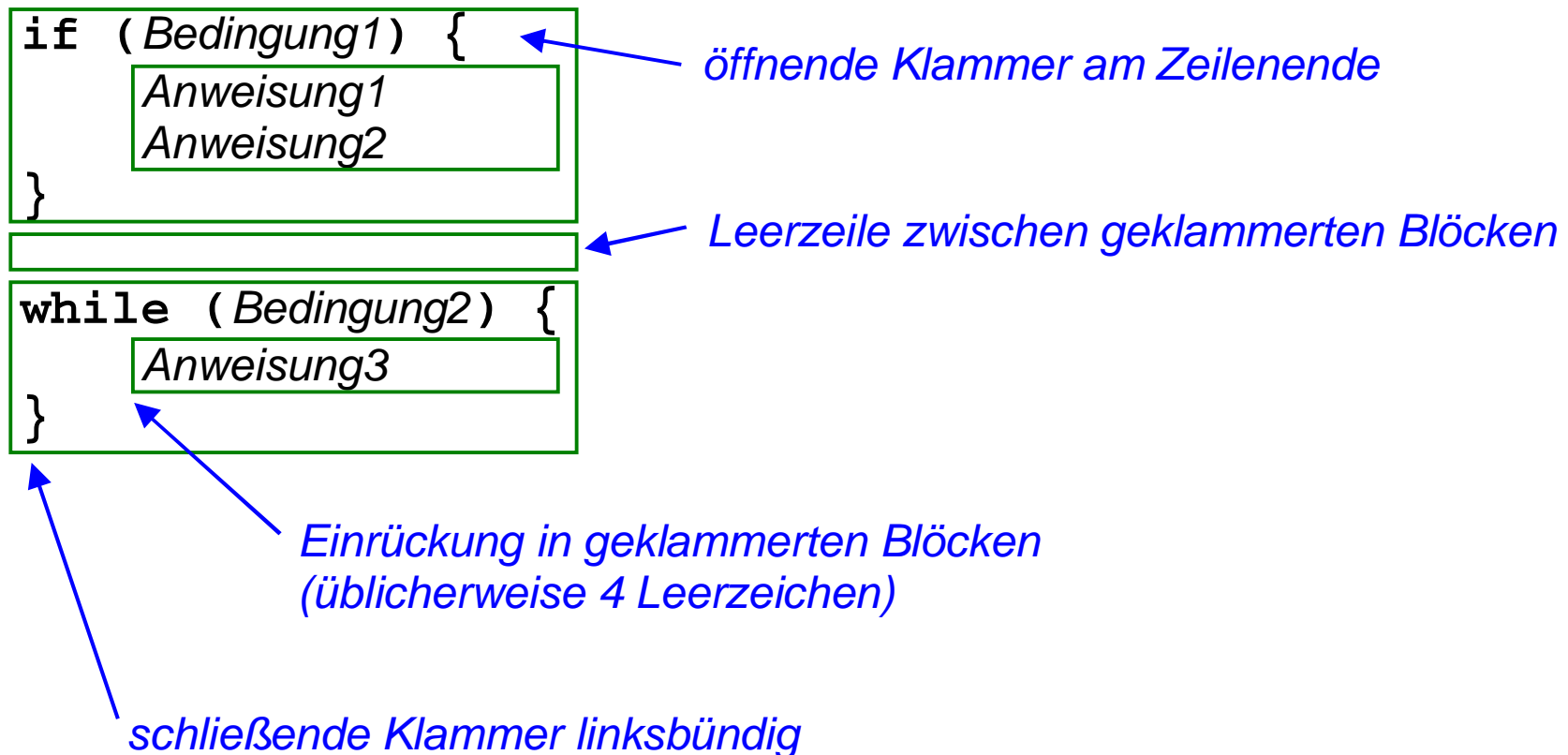
```
if (Bedingung) {  
    Anweisung;  
}
```

~~`if (Bedingung) Anweisung;`~~

Vereinfacht erheblich die Fehlersuche und Qualitätssicherung mit Werkzeugen wie Compiler, Debugger usw.

Java Anweisungen: Empfehlungen (2)

- Durch Zwischenraum (*Whitespace*), Klammerung und Einrückung die **Blockstruktur** der Ablaufsteuerung verdeutlichen:



Java Anweisungen: Index

Ablaufsteuerung 3-1
Anweisung 3-1
Anweisungsblock 3-1
Arithmetik 3-9,3-10
Assoziativität 3-3
Ausdruck 3-2
Ausnahmebehandlung 3-29,3-30,3-31
Auswertungsbaum 3-15
Bedingungsoperator 3-13
break 3-18 bis 3-23
case 3-18 bis 3-23
catch 3-29,3-31
continue 3-33,3-36
default 3-18 bis 3-23
Dekrement 3-8
do 3-24,3-27
else 3-16,3-17
Expression 3-2
Fallunterscheidung 3-18 bis 3-23
finally 3-29,3-31
for 3-25,28
for-each 3-25,28
if 3-16,3-17
Inkrement 3-8
links-assoziativ 3-3
Logik 3-11
Operator 3-3
Precedence 3-3
rechts-assoziativ 3-3
return 3-34,3-36
Schleife 3-24,3-25
Sprung 3-32 bis 3-36
Statement 3-1
Stelligkeit 3-3
switch 3-18 bis 3-23
throw 3-35,3-30,3-31
try 3-29,3-30,3-35
Typanpassung 3-14
Vergleich 3-12
Verzweigung 3-16,3-17
Vorrang 3-3
while 3-24,3-26,3-27
Zuweisung 3-7