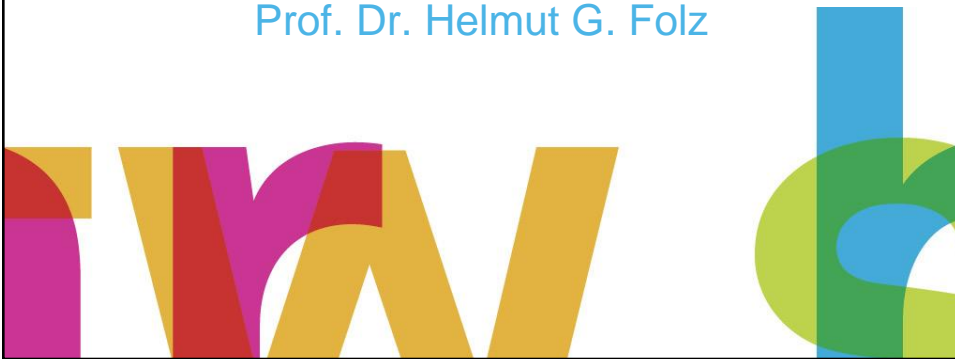


Kontrollstrukturen

Prof. Dr. Helmut G. Folz

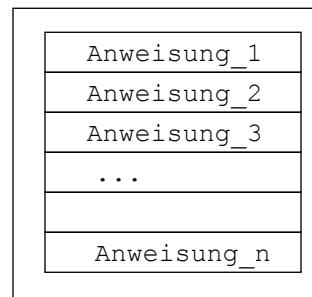


Blöcke

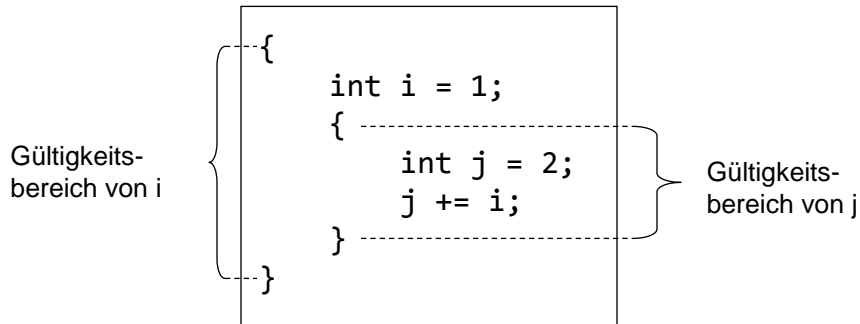
Eine **zusammengesetzte Anweisung** fasst mehrere Anweisungen zu einer Einheit, einem sogenannten **Block**, zusammen.

Syntax:

```
{  
    Anweisung_1  
    Anweisung_2  
    Anweisung_3  
    ...  
    Anweisung_n  
}
```



Blöcke und Gültigkeitsbereiche



Lebensdauer und Geltungsbereiche von Attributen

• (Objekt-)Attribute

- ⇒ Werden vom Konstruktor erzeugt,
- ⇒ werden automatisch initialisiert
- ⇒ sind in allen Methoden einer Klasse zugreifbar,
- ⇒ können je nach Sichtbarkeit auch von außerhalb der Klasse zugegriffen werden und
- ⇒ existieren so lange wie das zugehörige Objekt existiert

• Klassenattribute

- ⇒ Werden erzeugt sobald die Klasse geladen wird,
- ⇒ werden automatisch initialisiert,
- ⇒ sind in allen Methoden einer Klasse zugreifbar,
- ⇒ können je nach Sichtbarkeit auch von außerhalb der Klasse zugegriffen werden und
- ⇒ existieren so lange wie die Klasse geladen bleibt

Lebensdauer und Geltungsbereich von lokalen Variablen

- **Lokale Variablen**

- ⇒ Sind nur in den Block sichtbar, in dem sie definiert wurden,
- ⇒ werden **nicht** automatisch initialisiert (der Compiler überprüft aber ob nicht initialisierte lokale Variablen verwendet werden!)
- ⇒ werden mit dem Ende des Blocks wieder entfernt,
- ⇒ sind von außerhalb des Blocks, in dem sie definiert sind, nicht zugreifbar.

Die if-Anweisung

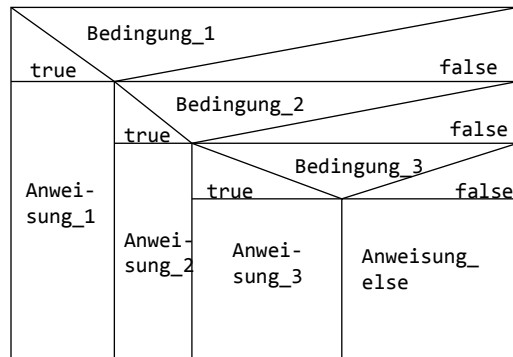
<pre>if (Bedingung) Ja-Anweisung else Nein-Anweisung</pre>	<pre>if (Bedingung) Ja-Anweisung</pre>
--	--

Mehrfachverzeigung mit if

Syntax:

```

if (Bedingung_1)
    Anweisung_1
else if (Bedingung_2)
    Anweisung_2
...
else if (Bedingung_n)
    Anweisung_n
else
    Anweisung_else
  
```



Beispiel: Notenrechner (1)

```

public class Notenrechner {
    private Scanner input = new Scanner(System.in);

    public void start() {
        char antwort = 'j';
        int punkte;
        double note;
        while (antwort == 'j') {
            System.out.print("Punkte eingeben : ");
            punkte = input.nextInt();
            note = berechneNote(punkte);

            if (note > 0.0)
                System.out.println("Note: " + note);
            else
                System.out.println(
                    "Ungültige Punktzahl eingegeben!");
            System.out.print("Noch einmal (j/n)? ");
            antwort = input.next().charAt(0);
        }
    }
}
  
```

Beispiel: Notenrechner (2)

```
public double berechneNote(int punkte) {
    double note;
    if (punkte >= 0 && punkte < 8)
        note = 5.0;
    else if (punkte >= 8 && punkte <= 10)
        note = 4.0;
    else if (punkte == 11)
        note = 3.7;
    else if (punkte == 12)
        note = 3.3;
    else if (punkte == 13)
        note = 3.0;
    else if (punkte == 14)
        note = 2.7;
    else if (punkte == 15)
        note = 2.3;
    else if (punkte == 16)
        note = 2.0;
```

Beispiel: Notenrechner (3)

```
    else if (punkte == 17)
        note = 1.7;
    else if (punkte == 18)
        note = 1.3;
    else if (punkte == 19 || punkte == 20)
        note = 1.0;
    else
        note = 0.0;
    return note;
}

public static void main(String[] args) {
    new Notenrechner().start();
}
```

Die switch-Anweisung

Syntax:

```
switch (Ausdruck) {
    case const1 : Anweisungen
                break;
    case const2 : Anweisungen
                break;
    case const3 : Anweisungen
                break;
    ...
    default      : ErsatzAnweisungen;
}
```

Die switch-Anweisung

- Der Ausdruck wird ausgewertet und muss ein Ergebnis vom Typ char, byte, short, int, Enum oder String haben.
- Die case-Konstanten müssen konstante Ausdrücke vom gleichen Typ wie der switch-Ausdruck sein (Literele oder final-Konstanten).
- Dieses Ergebnis wird mit den case-Konstanten *const1*, *const2*, .. verglichen, die zum Einsprung an die richtige Stelle dienen.
- Bei Übereinstimmung werden die zur passenden Konstante gehörigen Anweisungen ausgeführt.
- Die Angabe von break dient dazu, die switch-Anweisung zu verlassen. Fehlt die Angabe von break, so wird mit der nächsten Anweisungsfolge weitergemacht.
- Die nach default stehenden Anweisungen werden immer dann ausgeführt, wenn der switch-Ausdruck einen Wert liefert, der mit keiner der case-Konstanten übereinstimmt.

Die switch-Anweisung

Ausdruck					
const1	const2	const3	const4	sonst	
Anw	Anw	Anw	Anw	Anw	Anw

Beispiel: Notenrechner2 (1)

```

public class Notenrechner2 {
    public void start() {
        ...
    }

    public double berechneNote(int punkte) {
        double note;
        if (punkte >= 0 && punkte < 8)
            note = 5.0;
        else if (punkte >= 8 && punkte <= 10)
            note = 4.0;
        else {
            switch (punkte) {
                case 11: note = 3.7;
                        break;
                case 12: note = 3.3;
                        break;
                case 13: note = 3.0;
                        break;
            }
        }
    }
}

```

Beispiel: Notenrechner2 (2)

```

        case 14: note = 2.7;
                break;
        case 15: note = 2.3;
                break;
        case 16: note = 2.0;
                break;
        case 17: note = 1.7;
                break;
        case 18: note = 1.3;
                break;
        case 19:
        case 20: note = 1.0;
                break;
        default: note = 0.0;
    }
}
return note;
}

```

Beispiel: switch mit String-Objekten

```

public String typDesWochentags(String wochentag) {
    String typDesTags;
    switch (wochentag) {
        case "Montag":    typDesTags = "Start der Arbeitswoche";
                          break;

        case "Dienstag":
        case "Mittwoch":
        case "Donnerstag": typDesTags = "Wochenmitte";
                          break;

        case "Freitag":   typDesTags = "Ende der Arbeitswoche";
                          break;

        case "Samstag":
        case "Sonntag":   typDesTags = "Wochenende";
                          break;

        default:          throw new IllegalArgumentException(
                          "Falscher Wochentag: " + wochentag);
    }
    return typDesTags;
}

```


Die for-Anweisung

Syntax

```
for (Initialisierung; Schleifenbedingung;
      Iterationsanweisung )
    Wiederholungsanweisung
```

Ablauf:

1. Durchführung der Initialisierung (genau einmal)
2. Prüfung der Schleifenbedingung.
Wenn die Schleifenbedingung den Wert true hat:
 1. Wiederholungsanweisungen durchführen
 2. Iterationsanweisung durchführen
 Ansonsten Schleife beenden
3. Zurück zur Schleifenbedingung

Struktogrammdarstellung

Für i von 0 bis 99

Gebe i aus

```
// die ersten 100 Zahlen ausgeben
for (i = 0; i < 100; ++i)
    System.out.println(i);
```

Äquivalenz zur while-Anweisung

- Die for-Anweisung ist äquivalent zur folgenden while-Schleife:

```
Initialisierung
while ( Schleifenbedingung ) {
    Wiederholungsanweisungen
    Iterationsanweisung
}
```

```
// die ersten 100 Zahlen ausgeben
i = 0;
while ( i < 100 ) {
    System.out.println(i);
    i++;
}
```

Voreinstellungen

- Alle drei Ausdrücke im Kopf der Schleife sind optional, d.h. sie können weggelassen werden, nicht aber die trennenden Semikolons.
- Voreinstellungen sind:
 - Initialisierung: nichts
 - Schleifenbedingung: true
 - Iterationsanweisung: nichts

```
// Endlosschleife
for ( ; ; ) {
    ...
}
```

Beispiel: Test von for-Schleifen (1)

```
public class ForTest1 {
    public void start() {

        // Gerade Zahlen ausgeben
        for (int i = 0; i <= 20; i += 2)
            System.out.print(i + "\t");
        System.out.println();

        // Rückwärts zählen
        for (int i = 10; i > 0; i--)
            System.out.print(i + "\t");
        System.out.println();

        // Unterschied zu vorher??
        for (int i = 10; i > 0; --i)
            System.out.print(i + "\t");
        System.out.println();

    }
}
```

Beispiel: Test von for-Schleifen (2)

```
public class ForTest2 {
    public void summe1() {
        // 1. Version
        int n = 100;
        int i, summe = 0;

        for (i = 1; i <= n; i++)
            summe += i;

        System.out.println("1. Version: " + summe);
    }
}
```

Beispiel: Test von for-Schleifen (3)

```
public void summe2() {  
    // 2. Version  
    int n = 100;  
    int summe = 0;  
  
    for (int i = 1; i <= n; i++)  
        summe += i;  
  
    System.out.println("2. Version: " + summe);  
}
```

Beispiel: Test von for-Schleifen (4)

```
public void summe3() {  
    // 3. Version  
    int n = 100;  
    int i, summe;  
  
    for (i = 1, summe = 0; i <= n; summe += i, i++);  
  
    System.out.println("3. Version: " + summe);  
}
```

Beispiel: Test von for-Schleifen (5)

```
public void gebeSinusAus() {  
    // double-Variable hochzählen  
    for (double x = 0.0; x < 1.0; x += 0.1)  
        System.out.println(x + "\t" + Math.sin(x));  
}
```

Beispiel: Das große Einmaleins

Für m von start bis ende

Für n von 1 bis 10

Gebe $m * n$ aus

Neue Zeile

Beispiel: Das große Einmaleins

```
public class EinMalEins {
    public void einmaleins(int start, int ende) {
        int m, n;
        for (m = start; m <= ende; m++)
        {
            for (n = 1; n <= 10; n++ )
            {
                System.out.print(m * n + "\t");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        new EinMalEins().einmaleins(10, 20);
    }
}
```

Beispiel: Ein Zahlendreieck (1)

```
public class Dreieck {

    public void dreieck(int bis) {
        int i, j;
        for (i = 0; i <= bis; i++) {
            for (j = 0; j <= i; j++) {
                System.out.print(j + " ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        new Dreieck().dreieck(15);
    }
}
```

Beispiel: Ein Zahlendreieck (2)

```

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

```

Die while-Anweisung

Syntax	Struktogramm-Darstellung
<pre>while (Bedingung) { Wiederholungsanweisungen }</pre>	<pre> graph TD Loop[] subgraph Loop Bedingung[Bedingung] Wiederholungsanweisungen[Wiederholungsanweisungen] end Loop </pre>

Größter gemeinsamer Teiler

- Beispiel: Größter gemeinsamer Teiler zweier natürlicher Zahlen m und n

```
public long ggTintuitiv(long m, long n) {
    m = Math.abs(m);
    n = Math.abs(n);
    long teiler = 1;
    long groessterTeiler = 1;
    while (teiler <= m && teiler <= n) {
        if ( (m % teiler == 0) && (n % teiler == 0) )
            groessterTeiler = teiler;
        teiler++;
    }
    return groessterTeiler;
}
```

ggT intuitiv: Laufzeitmessung

m	n	ggT (m,n)	Laufzeit Java in ms	Laufzeit C++ in ms
123456	234567	3	0,36	0,44
1234567	2345678	1	3,30	3,80
12345678	23456789	1	32,60	37,60
123456789	234567891	9	327,40	375,33
1234567891	2147483647	1	3825,00	3802,67

(gemessen mit JDK 1.8 und GNU C++ 5.2.1 Intel Core i7-4900MQ mit 2,8 Ghz)

ggT intuitiv: Laufzeitmessung

- Durchschnittliche Laufzeit etwa:

$$3,0 \cdot 10^{-9} \cdot \min(m, n) \text{ sec}$$

- Hochgerechnet auf eine 18-stellige Zahl

$$3,0 \cdot 10^{-9} \cdot 10^{18} \text{ sec}$$

$$= 3,0 \cdot 10^9 \text{ sec}$$

$$\approx 95 \text{ Jahre}$$

Der Euklidische Algorithmus

n und m positiv eingeben

$$r = m \bmod n$$

wiederhole, solange $r > 0$

$$m = n$$

$$n = r$$

$$r = m \bmod n$$

Der ggT steht in n

Der Euklidische Algorithmus

```
public long ggT (long m, long n) {
    m = Math.abs(m);
    n = Math.abs(n);
    long r = m % n;
    while (r > 0) {
        m = n;
        n = r;
        r = m % n;
    }
    return n;
}
```

Laufzeitmessung

```
public void messeGGTintuitiv(int zahl1, int zahl2,
                             int anzahl) {
    long zeit = System.currentTimeMillis();
    int wert = 0;
    for (int i = 1; i <= anzahl; i++)
        wert = ggTintuitiv(zahl1, zahl2);

    double dzeit = (double)(System.currentTimeMillis()
                             - zeit)/anzahl;
    System.out.printf("ggT(%18d, %18d) = %2d",
                      zahl1, zahl2, wert);
    System.out.printf(" Zeit: %8.2f ms Faktor: %f\n",
                      dzeit, dzeit /zahl1);
}
```

Laufzeitmessung

```
public void messeGGT(long zahl1, long zahl2) {
    int anzahl = 200000;
    long zeitMilli = System.currentTimeMillis();
    long ggT = 0L;

    for (int i = 0; i < anzahl; i++)
        ggT = ggT(zahl1, zahl2);

    zeitMilli = System.currentTimeMillis() - zeitMilli;
    System.out.printf("ggT(%18d, %18d) = %2d",
        zahl1, zahl2, ggT);
    System.out.printf(" Zeit: %5d ms Faktor: %f\n",
        zeitMilli, (double)zeitMilli /
            Math.Log((double)zahl1));
}
```

Euklid: Laufzeitmessung

m	n	ggT (m,n)	Laufzeit für 200000 Aufrufe Java in ms	Laufzeit für 200000 Aufrufe C++ in ms
123456	234567	3	14	22
1234567	2345678	1	18	24
12345678	23456789	1	15	20
123456789	234567891	9	16	23
1234567891	2147483647	1	48	62
123456789012345678	234567890123456789	1	57	73

(gemessen mit JDK 1.8 und GNU C++ 4.8.1 Intel Core i7-4900MQ mit 2,8 Ghz)

Euklid: Laufzeitmessung

- Durchschnittliche Laufzeit je ggT etwa:

$$1,0 \cdot 10^{-8} \cdot \log(\min(m, n)) \text{ sec}$$

- Hochgerechnet auf eine 18-stellige Zahl

$$1,0 \cdot 10^{-8} \cdot 41,44 \cdot \text{sec}$$

$$\approx 0,0004 \text{ m sec}$$

Die do-Anweisung

Syntax	Struktogramm-Darstellung
<pre>do { Wiederholungsanweisungen } while (Bedingung);</pre>	

Beispiel: do .. while

```

public boolean weitermachen() {
    char antwort;
    do {
        System.out.print("Noch einmal (j/n)? ");
        antwort = input.next.charAt(0);
    } while (antwort != 'j' && antwort != 'n');

    return antwort == 'j';
}

public void start() {
    ...
    do {
        ...
    } while (weitermachen());
}

```

Äquivalenz zwischen do- und while-Schleife

while-Schleife	äquivalente do-Schleife
<pre> while(Schleifenbedingung) { Wiederholungsanweisungen } </pre>	<pre> if(Schleifenbedingung) do { Wiederholungsanweisungen } while(Schleifenbedingung); </pre>
do-Schleife	äquivalente while-Schleife
<pre> do { Wiederholungsanweisungen } while(Schleifenbedingung); </pre>	<pre> Wiederholungsanweisungen; while(Schleifenbedingung) { Wiederholungsanweisungen } </pre>

Die break-Anweisung

```

while (Schleifenbedingung) {
    ...
    ...
    if (Fehlerbedingung)
        break; // Sprung zur ersten Anweisung
                ! // hinter der while-Schleife
    ... !
} !
<-----+

```

Beispiel: break

```

for (...; ...;...) {
    while (Schleifenbedingung) {
        ...
        ...
        if (Fehlerbedingung)
            break; // Sprung zur ersten Anweisung
                    ! // hinter der while-Schleife
        ... !
    } !
    ... <-----+
}

```

Beispiel: Messwerte summieren (1)

```
double summe = 0, messwert;  
  
messwert = input.nextDouble();  
for(n = 0; (n < 100) && (messwert >= 0); n++)  
{  
    if (messwert >= 0)  
        summe += messwert;  
    messwert = input.nextDouble();  
}
```

Beispiel: Messwerte summieren (2)

```
double summe = 0, messwert;  
  
for(n = 0; n < 100; n++)  
{  
    messwert = input.nextDouble();  
    if (messwert < 0)  
        break; // Absprung hinter die Schleife  
    summe += messwert;  
}
```

Die continue-Anweisung

```

while (Schleifenbedingung) {
    ...
    ...
    if (Bedingung)
        continue;    // Sprung ans Ende des
                       // Schleifenrumpfes
    ...
} <-----+

```

Beispiel: continue

```

for (...; ...; ...) {
    while (Schleifenbedingung) {
        ...
        ...
        if (Bedingung)
            continue; // Sprung ans Ende der
                       // while-Schleife
        ...
    } <-----+
    ...
}

```


Beispiel: break mit Marke

```
// break mit einer Marke anwenden
public static void main(String[] args) {
    args:
    while (true) {
        System.out.println(args[0]);
        if (Math.random() > 0.4)
            break args;    // Die Kontrolle geht zur
                           // Anweisung hinter der
                           // markierten while-Anweisung
        System.out.println("Noch einmal!");
    }
    System.out.println("Fertig!");    // <----
}
```

Beispiel: break mit Marke

```
// Sprung aus einer inneren Schleife

Marke: while (Bedingung) {                // while 1
    while (Bedingung) {                    // while 2
        ...
        break Marke;
        // Sprung hinter das Ende
        // der markierten Schleife
    }
    ...
} // Ende while 2
} // Ende while 1
```

Beispiel: continue mit Marke

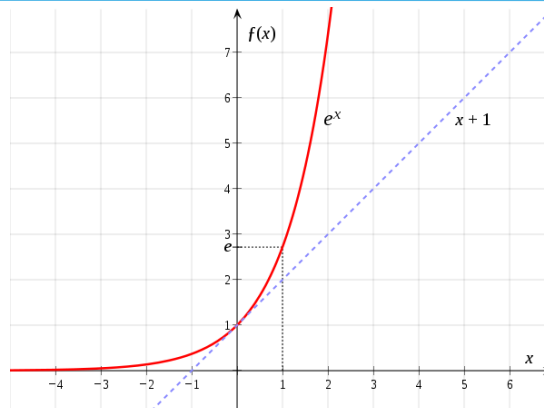
```
// Analoge Anwendungen für continue  
Marke2: for (i = 0; i < imax; i++) {    // for 1  
    for (j = 0; j < jmax; j++) { // for 2  
        ...  
        if (Bedingung)  
            continue Marke2;  
    }    // Ende for 2  
} // Ende for 1
```

Iterationen

Iteration

- mehrmaliges Durchlaufen der selben Anweisungskette.
- oft in Verbindung mit der Tatsache, dass Werte der i-ten Ausführung von Werten der (i-1)-ten (und weiteren vorangegangenen) Ausführungen abhängen.

Die Exponentialfunktion



$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \dots + \frac{x^n}{n!} + \dots$$

Exponentialfunktion: naiver Ansatz (1)

```
public class Exp1 {
    /** Berechnen der Exponentialfunktion */
    public double exp(double x) {
        double epsilon = 1e-10;
        double summand = 0.0;
        double wertAlt;
        double wert = 1.0;
        int i = 0;

        do {
            i++;
            wertAlt = wert;
            summand = potenz(x, i) / fakultaet(i);
            wert += summand;
        } while (Math.abs(wert - wertAlt) > epsilon);

        ...
    }
}
```

Exponentialfunktion: naiver Ansatz (2)

```
/** potenz berechnet die Potenz x hoch n fuer positives
 * n
 */
public double potenz(double x, int n) {
    double produkt = 1;
    for (int i = 0; i < n; i++)
        produkt *= x;
    return produkt;
}

/** fakultaet berechnet die Fakultaet von n fuer
 * positives n
 */
public double fakultaet(int n) {
    double wert = 1.0;
    for (int i = 1; i <= n; i++)
        wert *= i;
    return wert;
}
```

Exponentialfunktion: naiver Ansatz (3)

Wieso ist diese Lösung schlecht?

Exponentialfunktion: besserer Ansatz

```
public class Exp2 {
    /** Berechnen der Exponentialfunktion */
    public double exp(double x) {
        double epsilon = 1e-10;
        double summand = 1.0;
        double wert = 1.0;
        double wertAlt;
        int i = 0;

        do {
            i++;
            wertAlt = wert;
            summand = summand * x / i;
            wert += summand;
        } while (abs(wert - wertAlt) > epsilon);
        ....
    }
}
```

Der Potenzierungsalgorithmus

Aufgabe:

Berechne x^n mit x double-Wert und n > 0 int-Wert.

Vorläufige Implementierung:

```
public double potenz(double x, int n) {
    double produkt = 1;
    for (int i = 0; i < n; i++)
        produkt *= x;
    return produkt;
}
```

Der Potenzierungsalgorithmus

- Betrachte die Binärdarstellung der Zahl n:

$$n = \sum_{i=1}^k \alpha_i 2^i = \alpha_0 2^0 + \alpha_1 2^1 + \dots + \alpha_k 2^k \quad \text{mit } \alpha_i = 0 \text{ oder } 1$$

- Nun gilt:

$$x^n = x^{\sum_{i=1}^k \alpha_i 2^i} = \prod_{i=1}^k x^{\alpha_i 2^i} = x^{\alpha_0 2^0} \cdot x^{\alpha_1 2^1} \cdot \dots \cdot x^{\alpha_k 2^k}$$

Der Potenzierungsalgorithmus

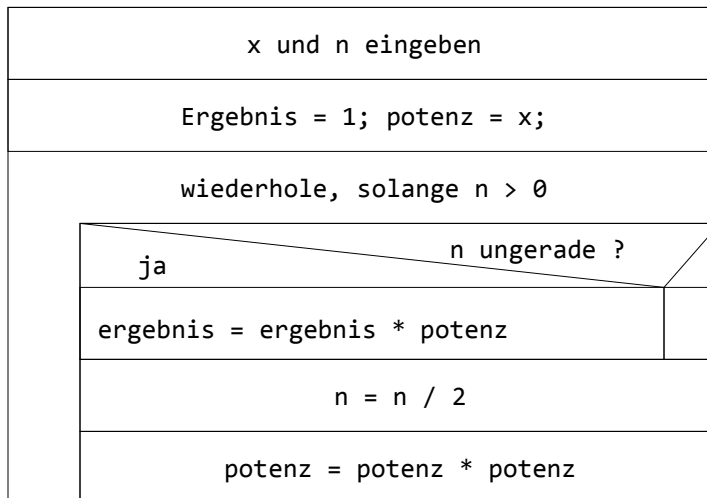
- Beispiel: n = 23

$$23 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4$$

- Nun gilt:

$$x^{23} = x^{1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4} = x^{2^0} \cdot x^{2^1} \cdot x^{2^2} \cdot x^{0 \cdot 2^3} \cdot x^{2^4}$$

Potenzierungsalgorithmus



Potenzierungsalgorithmus

```

public double binPotenz(double x, int n) {
    double ergebnis = 1.0;
    double potenz = x;
    while (n > 0) {
        if (n % 2 != 0)           // n ungerade?
            ergebnis = ergebnis * potenz;
        n = n / 2;
        potenz = potenz * potenz;
    }
    return ergebnis;
}

```