

Inhalt

4	Anweisungen	4-2
4.1	<i>Strukturierte Programmierung</i>	4-2
4.1.1	Geschichte	4-2
4.1.2	Strukturierung im Kleinen	4-2
4.2	<i>Einige Beispielanwendungen</i>	4-4
4.2.1	Addierer (do-Schleife)	4-4
4.2.2	Ein- Mal- Eins (for-Schleife, if-Anweisung)	4-5
4.2.3	Lineare Gleichung (if-else-Anweisung)	4-6
4.2.4	Einfacher Rechner (switch-Anweisung)	4-8
4.2.5	Summieren (while-Schleife, break, continue)	4-10

4 Anweisungen

4.1 Strukturierte Programmierung

4.1.1 Geschichte

In der Anfangszeit des Programmierens, bis in die sechziger Jahre des 20. Jh. hinein, gab es *keine Programmiermethodik*, keine Techniken und keine Regeln für das fachmännische Schreiben größerer Programme. Jeder Programmierer hatte vielmehr seine eigene Vorgehensweise.

Dabei wurden zu dieser Zeit bereits enorm große Programme geschrieben. Die *Komplexität* dieser großen Programme wuchs ihren Entwicklern über den Kopf. Ein durchaus üblicher Programmquelltext von 100'000 Anweisungen ergibt einen Programmtext von 2'000 Seiten, wenn man eine Anweisung pro Zeile und die Seite mit 50 Zeilen ansetzt. Dabei werden erläuternde Kommentare, die bei der Größe solch eines Programms unabdingbar sind, noch nicht einmal berücksichtigt. Die *Fehlerhäufigkeit* war groß. Bei der Beseitigung der gefundenen Fehler schlichen sich neue ein.

⇒ **Chaos**

Ende der sechziger Jahre fanden zwei von der NATO ausgerichtete Tagungen zu diesem Thema statt:

- 1968 in Garmisch, „*Working Conference on Software Engineering*“
- 1969 in Rom, „*Working Conference on Software Engineering Techniques*“

Auf ihnen wurden erstmalig die Probleme der Entwicklung großer Programme explizit benannt und diskutiert. Man stellte fest, dass Software das Ergebnis von **Ingenieurtätigkeit** ist und, wie jedes *industrielle Produkt*, der methodischen *Planung, Entwicklung, Herstellung* und *Wartung* bedarf. So entstand ein neues Wissenschaftsgebiet, welches heute unter dem Begriff **Software Engineering** bzw. **Softwaretechnik** läuft.

4.1.2 Strukturierung im Kleinen

Die Gestaltung der **Ablaufstruktur** von Algorithmen in der Programmierung lässt sich auf *zwei elementare Konstruktionen* zurückführen:

1. **Einfache Aktionen** (Wertzuweisungen, Prozeduraufrufe, ...)
2. **Binäre Verzweigungen zur Ablaufsteuerung** (bedingte Sprünge)

Viele Probleme bei der Programmierung konnten auf die „Freiheit“ der Verwendung von Verzweigungen zurückgeführt werden. Deshalb war man bemüht, diese einzuschränken. **EDSGER W. DIJKSTRA** [1930-2002], ein niederländischer Informatiker, führte 1968 in einem Artikel „*Go To Statement Considered Harmful*“¹ aus: „*die Qualität eines Programms sei umgekehrt proportional zu der Anzahl der darin enthaltenen goto - Sprünge*“. Er schlug deshalb vor, dass man sich auf 5 Muster in den Sprachkonstrukten beschränken soll, die sogenannten **D-Diagramme**.

¹ <http://www.acm.org/classics/oct95/>

Dijkstra-Diagramm (D-Diagramm)

1. Eine *einfache Aktion* ist ein *D-Diagramm*.
2. Wenn *A* und *B* *D-Diagramme* sind, so auch
 - a. *A B*
 - b. *if condition then A end*
if condition then A else B end
 - c. *while condition do A end*
3. Nichts sonst ist ein *D-Diagramm*.

Wertzuweisung

**Anweisungssequenz
Auswahanweisungen**

Schleifenanweisung

Damit war der Anfang der modernen Softwareentwicklung markiert. Statt wilder Sprünge durch *goto*-Befehle sollten *if* und *while* den Programmfluss strukturieren. Die Beschränkung auf *D-Diagramme* wird als **goto-lose-Programmierung** bezeichnet. Fortan galten Programme mit *goto* als sogenannte „Spaghetti-Programme“ *verwerflich*.

In den modernen Programmiersprachen findet man heute *mehrere* Formen von **Auswahl- und Schleifenanweisungen**. Hinzu sind auch noch einige *explizite strukturbezogene Sprunganweisungen* gekommen. Diese Erweiterung zur Ablaufsteuerung sieht man als *unbedenklich* an.

Unverkennbar erfüllt unser Modellrechner die Bedingungen von Dijkstra.

Java-Anweisungen:

**Ausdrucksanweisungen
zusammengesetzte Anweisungen
Schleifenanweisungen
Auswahanweisungen
strukturbezogene Sprunganweisungen**

Einfache Aktionen

Ausdrucksanweisungen

Ausdruck;

Ablaufsteuerung

zusammengesetzte Anweisungen

{ Anweisung Anweisung ... Anweisung }

Auswahanweisungen

if(Bedingung) Anweisung

if(Bedingung) Anweisung else Anweisung

switch(Ausdruck)

{

case Konstante: Anweisung Anweisung ...

case Konstante: Anweisung Anweisung ...

...

default: Anweisung Anweisung ...

}

Schleifenanweisungen

while(Bedingung) Anweisung

do Anweisung while(Bedingung);

for(Ausdruck 1; Ausdruck 2; Ausdruck 3) Anweisung

strukturbezogene Sprunganweisungen

continue; break; return;

4.2 Einige Beispielanwendungen

4.2.1 Addierer (do-Schleife)

Wiederholtes Summieren von zwei Dezimalzahlen: Zunächst wird aus der Aufgabenstellung die **Grobstrukturierung** unter Verwendung des *EVA-Prinzips* entwickelt.

Kritische Testbeispiele:

$2e308 + 1 \Rightarrow \text{Infinity}$, $1e16 + 1 \Rightarrow 1e16$, $1e-16 + 1 \Rightarrow 1$, $1e-16 + -1 \Rightarrow -0.9999999999999999$.

EinfacherAddierer.java (Grobstruktur)

```
public class EinfacherAddierer
{
    public static void main( String[] args)
    {
        char weiter;                // j oder n

        do
        {
            // Eingabe der Summanden
            // Berechnen der Summe
            // Ausgabe der Summe
            // Weiter
        } while( weiter == 'j');
    }
}
```

Ein Programm entsteht aus der Grobstruktur durch **schrittweise Verfeinerung**.

EinfacherAddierer.java

```
// EinfacherAddierer.java                MM 2014
import Tools.IO.*;                       // Eingaben

/**
 * Einfacher Addierer,
 * addiert beliebig oft zwei Dezimalzahlen.
 */
public class EinfacherAddierer
{
    /**
     * Eingabe der Summanden,
     * Berechnen und Ausgabe der Summe;
     * Abbruch auf Wunsch des Nutzers.
     */
    public static void main( String[] args)
    {
        char weiter;                // j oder n

        do
        {
            // Eingabe der Summanden
            double summand1
```

```

    = IOTools.readDouble( "Summand1 = ");
    double summand2
    = IOTools.readDouble( "Summand2 = ");

    // Berechnen der Summe
    double summe = summand1 + summand2;

    // Ausgabe der Summe
    System.out.println
    ( summand1 + " + " + summand2 + " = " + summe);

    // Weiter
    weiter = IOTools.readChar( "Weiter(j/n)? ");
} while( weiter == 'j');

System.out.println( "Programm beendet");
}
}

```

4.2.2 Ein- Mal- Eins (for-Schleife, if-Anweisung)

Das kleine Einmaleins mit strukturierter Ausgabe (Es wird berücksichtigt, dass es ein-, zwei- und dreistellige Produkte gibt.):

EinMalEins.java (Grobstruktur)

```

public class EinMalEins
{
    public static void main( String[] args)
    {
        // Zeile
        for( int z = 1; z <= 10; z++)
        {
            // Spalte in der Zeile
            for( int s = 1; s <= 10; s++)
            {
                // Abstand
                // Produkt
            }
            // Zeilenvorschub
        }
    }
}

```

EinMalEins.java

```
//EinMalEins.java
```

MM 2014

```

/**
 * Das kleine 1 x 1.
 */
public class EinMalEins

```

```

{
/**
 * Zeilenweise Ausgabe des kleinen 1x1.
 */
public static void main( String[] args)
{
    // Zeile
    for( int z = 1; z <= 10; z++)
    {
        // Spalte in der Zeile
        for( int s = 1; s <= 10; s++)
        {
            // Abstand
            if( z * s < 10) System.out.print( " ");
            if( z * s < 100) System.out.print( " ");

            // Produkt
            System.out.print( " " + z * s);
        }
        // Zeilenvorschub
        System.out.println();
    }
}
}

```

EinMalEins.out

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100

4.2.3 Lineare Gleichung (if-else-Anweisung)

Lösen der Gleichung $ax+b=0$.

Testbeispiele: $3x+4=0$ ($L = \{-\frac{4}{3}\}$), $3x=0$ ($L = \{0\}$), $0x+2=0$ ($L = \{ \}$), $0x=0$ ($L = R$).

LinGleichung.java (Grobstruktur)

```

public class LinGleichung
{
    public static void main( String args[])
    {
        // Einlesen der Parameter a und b

```

```

// Berechnen und Ausgabe, Fallunterscheidung
    if( a == 0)
        if( b == 0)                // unendlich viele Loesungen
            else                    // keine Loesung
        else                        // 1 Loesung
    }
}

```

LinGleichung.java

```

//LinGleichung.java                                MM 2014

import Tools.IO.*;                                // Eingaben

/**
 * Berechnen der linearen Gleichung  $ax + b = 0$ .
 */
public class LinGleichung
{
    /**
     * Eingabe der Parameter a und b,
     * Berechnen und Ausgabe der Loesung.
     */
    public static void main( String[] args)
    {
        // Eingabe der Parameter a und b
        System.out.println( "Loesung  $ax + b = 0$ ");

        double a = IOTools.readDouble( "a = ");
        double b = IOTools.readDouble( "b = ");

        // Berechnen und Ausgabe, Fallunterscheidung
        if( a == 0)
            if( b == 0)                // unendlich viele Loesungen
                System.out.println("L = R");
            else                    // keine Loesung
                System.out.println("L = {}");
            else                        // 1 Loesung
                System.out.println("L = {" + -b/a + "}");
        }
    }

    /* ----- */
    /*
    Testbeispiel  $3.0x + 4.0 = 0$ 
    L = {-1.3333333333333333}

    Testbeispiel  $3.0x + 0.0 = 0$ 
    L = {-0.0} !!!

    Testbeispiel  $0.0x + 2.0 = 0$ 
    L = {}
    */
}

```

```

Testbeispiel 0.0x + 0.0 = 0
L = R
*/

```

4.2.4 Einfacher Rechner (switch-Anweisung)

Rechner führt Grundrechenarten +, -, * und / aus: Operanden und Operator werden eingegeben, Ergebnis wird ausgegeben. Division durch Null wird abgefangen.

EinfacherRechner.java (Grobstruktur)

```

public class EinfacherRechner
{
    public static void main( String[] args)
    {
        char weiter;                // j oder n
        do
        {
            // Aufgabe
            // Berechnen und Ausgabe des Ergebnisses
            switch (op)
            {
                case '+':            // Addition
                    break;
                case '-':            // Subtraktion
                    break;
                case '*':            // Multiplikation
                    break;
                case '/':            // Division
                    break;
                default:
                    System.out.println( "Fehlerhafte Eingabe");
            }
            // Weiter
        } while( weiter == 'j');
    }
}

```

Einfacher Rechner.java

```

// EinfacherRechner.java                MM 2014
import Tools.IO.*;                       // Eingaben

/**
 * Einfacher Rechner,
 * fuehrt Grundrechenarten fuer Dezimalzahlen aus.
 */
public class EinfacherRechner
{
    /**
     * Eingabe der Aufgabe (Operand1 Operator Operand2),
     * Berechnen und Ausgabe des Ergebnisses;

```



```
* Abbruch auf Wunsch des Nutzers.
*/
public static void main( String[] args)
{
    char weiter; // j oder n
    do
    {
        // Aufgabe
        double operand1
        = IOTools.readDouble( "Operand1 = ");
        char op
        = IOTools.readChar( "Operator (+, -, *, /) ");
        double operand2
        = IOTools.readDouble( "Operand2 = ");

        // Berechnen und Ausgabe des Ergebnises
        double ergebnis;
        switch (op)
        {
            case '+': // Addition
                ergebnis = operand1 + operand2;
                System.out.print
                ( operand1 + " + " + operand2 + " = ");
                System.out.println( ergebnis);
                break;

            case '-': // Subtraktion
                ergebnis = operand1 - operand2;
                System.out.print
                ( operand1 + " - " + operand2 + " = ");
                System.out.println( ergebnis);
                break;

            case '*': // Multiplikation
                ergebnis = operand1 * operand2;
                System.out.print
                ( operand1 + " * " + operand2 + " = ");
                System.out.println( ergebnis);
                break;

            case '/': // Division
                if( operand2 != 0)
                {
                    ergebnis = operand1 / operand2;
                    System.out.print
                    ( operand1 + " / " + operand2 + " = ");
                    System.out.println( ergebnis);
                }
                else System.out.println( "Division durch 0");
                break;

            default:
```

```

        System.out.println( "Fehlerhafte Eingabe");
    }

    // Weiter
    weiter = IOTools.readChar( "Weiter(j/n)? ");
} while( weiter == 'j');

System.out.println( "Programm beendet");
}
}

```

4.2.5 Summieren (while-Schleife, break, continue)

Aufsummieren beliebig vieler natürlicher Zahlen.

Kritische Testbeispiele: $2\,000\,000\,000 + 150\,000\,000 \Rightarrow -2 \dots$

EndlosSchleifen.java

```

//EndlosSchleifen.java MM 2014

import Tools.IO.*; // Eingaben

/**
 * Aufsummieren beliebig vieler natuerlicher Zahlen.
 */
public class EndlosSchleifen
{
    /**
     * Liest natuerliche Zahlen ein und addiert diese,
     * Abbruch mit -1.
     */
    public static void main( String[] args)
    {
        int zahl = 0, summe = 0;

        while( true) // Formale Endlosschleife
        {
            zahl = IOTools.readInteger
            ( "Naechste natuerliche Zahl (Abbruch mit -1): ");

            if( zahl < 0) break; // Schleifenabbruch
            if( zahl == 0) continue; // Durchlaufabbruch

            summe += zahl;
        }

        System.out.println( "\nSumme = " + summe);
    }
}

```