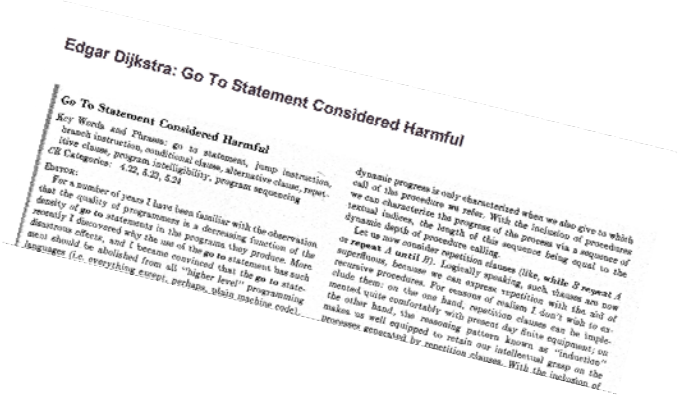
DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

Vorlesung Programmieren
Thema 4: Kontrollstrukturen

Olaf Herden

Fakultät Technik

Studiengang Informatik



Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

for

switch

if

while

Stand: 11/2023

Gliederung

- Elementare Kontrollstrukturen
- Bedingte Anweisung
- Schleifen
- Sprünge
- Unterprogramme

Deklaration (I)

- Anlegen Variablen bestimmten Typs:
  - Reservierung Speicherplatz
  - Initialisierung
- Syntax:

```
<Deklaration> ::= <Typ>
                  <Bezeichner> ["=" <Ausdruck> ]
                  {"," <Bezeichner> ["=" <Ausdruck> ] }
                  ";"
```

- Beispiele:

```
(1) int i = 7;
(2) int j;
(3) int k = 5, l, m, n = 9;
```

Deklaration (II)

- Bedingungen:
  - Variable und Initialisierungsausdruck müssen typkonform sind, z.B. `int i = 3.14;` ist verboten
  - Variablenname darf nicht im eigenen Initialisierungsausdruck vorkommen, z.B. `int i = i + 6;` ist verboten

Ausdruck

- Syntax: `<Ausdrucksanweisung> ::= <Ausdruck> ";"`
- `<Ausdruck>` muss Zuweisungsausdruck oder Funktionsaufruf sein
- Semantik: Berechnung des Ausdrucks
- Beispiel:

```
(1) float pi = 3.1415;
(2) float radius = 7.2;
(3) float umfang = 2 * pi * radius;
```

(Zuweisungs)ausdruck

- Anm.: Funktionsaufrufe werden später behandelt

Zuweisung

- Syntax:

  ```
  <Zuweisungsausdruck> ::= <Variablenname> "=" <Ausdruck>
  <Zuweisung> ::= <Zuweisungsausdruck> ";"
  ```

- Semantik:
  - Berechnung des Ausdrucks
  - Zuweisung des berechneten Wertes an die Variable
- Beispiele:

```
(1) float i = 3.;
(2) float j = 4.5;
(3) float k;
(4) k = i * j;
```

  Resultat: k = 13.5

- Bedingungen:
  - Variablenname muss gültig sein, d.h. vorher deklariert
  - Typ des Ausdrucks und der Variable müssen kompatibel sein

Block

- Syntax: `<Block> ::= "{" { <Anweisung> } "}„`
  `<Anweisung> ::= (<Elementare_Anweisung> | <Block>)`
- Semantik:
  - Keine Auswirkungen auf den Programmablauf
  - Dient lediglich der Strukturierung
  - Innerhalb eines Blockes können eigene Variable deklariert werden
- Beispiel:

```
(1) {
(2)   int i = 5;
(3)   {
(4)     int j = 8;
(5)     i = j;
(6)   }
(7)    i++;
(8)    j++;
(9) }
```

(Innerer) Block

(Äußerer) Block

Fehler: Variable j im äußeren Block nicht gültig

Gültigkeitsbereich und Lebensdauer

- Gültigkeitsbereich einer Variablen:
  - Zur Compilezeit relevant
  - Teil des Programms, in dem Variable über ihren Namen angesprochen werden kann
  - Gültigkeitsbereich einer Variablen:
    - Block der Definition
    - Alle inneren Blöcke
  - Variablennamen müssen innerhalb eines Blocks und aller inneren Blöcke eindeutig sein
- Lebensdauer einer Variablen:
  - Zur Laufzeit relevant
  - Lebensdauer einer Variablen beginnt mit dem Anlegen
  - Lebensdauer einer Variablen endet mit dem Verlassen des Blocks

Sequenz

- Syntax: `<Sequenz> ::= <Anweisung_1> ";" <Anweisung_2> ";"`
- Semantik: Durch Semikolon getrennte Anweisungen werden hintereinander ausgeführt
- Beispiel:

```
(1) int i = 3;
(2) int j = 5;
(3) j = j + i;
(4) i -= 8;
```

Resultat: i = -5 und j = 8

Übersicht

- Elementare Kontrollstrukturen
- Bedingte Anweisung
- Schleifen
- Sprünge
- Unterprogramme

Bedingte Anweisung (I)

- Syntax: `<if-Anweisung> ::= "if (" <Boolescher_Ausdruck> ")" <Anweisung_1> ["else" <Anweisung_2>]`
- Semantik:
  - Auswertung des Ausdrucks
  - Ist dieser wahr, so wird Anweisung_1 ausgeführt
  - Sonst wird Anweisung_2 ausgeführt
- Beispiel:

```
(1) if (i % 2 == 0)
(2)    System.out.println("Zahl ist gerade.");
(3) else
(4)    System.out.println("Zahl ist ungerade.");
```

Bedingte Anweisung (II)

- Häufiger Fehler:

```
(1) if (i % 2 == 0)
(2)   System.out.print("Zahl ist ");
(3)   System.out.println("gerade.");
```

führt bei ungeradem i zur Ausgabe `gerade.`

- Sollen beide Anweisungen nur bei geradem i ausgeführt werden, müssen diese zu einem Block zusammengefasst werden:

```
(1) if (i % 2 == 0){
(2)   System.out.print("Zahl ist ");
(3)   System.out.println("gerade.");
(4) }
```

Bedingte Anweisung (III)

- Erweiterung um mehrere Möglichkeiten:

```
<if-Anweisung> ::= "if (" <Boolescher_Ausdruck> ")"
                   <Anweisung_1>
                     {"else if" <Boolescher_Ausdruck>
                       <Anweisung_2>}
                   ["else" <Anweisung_3>]
```

- Beispiel:

```
(1) if (i == 0)
(2)   System.out.println("Null.")
(3) else if (i == 1)
(4)   System.out.println("Eins.")
(5) else if (i == 2)
(6)   System.out.println("Zwei.")
(7) else System.out.println("Weder 0 noch 1 noch 2.");
```

Übersicht

- Elementare Kontrollstrukturen
- Bedingte Anweisung
- Schleifen
- Sprünge
- Unterprogramme

while-Anweisung

- Syntax:

```
<while-Anweisung> ::= "while (" <boolescher Ausdruck> ")"
                      (<Anweisung>)
```

- Semantik:
  - Boolescher Ausdruck wird berechnet
  - Wenn true ⇒ Anweisung ausführent
  - Sonst ⇒ Fortfahren mit nächster Anweisung
- Beispiel:

```
(1) int i = 1;
(2) int grenze = 20;
(3) while (i <= grenze){
(4)   System.out.println(i);
(5)   i++;
(6) }
```

do-while-Anweisung (I)

- Syntax:

```
<do-Anweisung> ::= "do"
                   <Anweisung>
                    "while (" <Boolescher Ausdruck> ");"
```

- Semantik:
  - Ausführung Anweisung
  - Berechnung Boolescher Ausdruck
    - Liefert true $\Rightarrow$ Erneutes Ausführen Anweisung
    - Liefert false $\Rightarrow$ Ausführen Anweisung nach `do-while`-Anweisung

do-while-Anweisung (II)

- Beispiel:

```
(1) int i = 0;
(2) int grenze = 20;
(3) do{
(4)    System.out.println(i);
(5)    i++;
(6) }
(7) while (i <= grenze);
```

`while` vs. `do-while`-Anweisung

- Unterschied der `do-while`- zur `while`-Anweisung:
  - Schleifenrumpf wird mindestens einmal ausgeführt
- Beispiele der letzten beiden Folien unterschiedliche Auswirkung
- Schleifentypen semantisch äquivalent:
  `<anweisung1> while (<bedingung>) <anweisung1>`
  gleichbedeutend mit
  `do <anweisung1> while (<bedingung>);`
- Benennung:
  - `while`-Anweisung wird manchmal als kopfgesteuerte Schleife bezeichnet
  - `do-while`-Anweisung wird manchmal als fußgesteuerte Schleife bezeichnet

`for`-Schleife (I)

- Syntax:

```
<for-Anweisung> ::= "for"
                    "(" [ <Init-Anweisung> ] ";"
                        [ <boolescher Ausdruck> ] ";"
                        [ <Inkrement-Ausdruck> ]
                    ")" <Anweisung>
```

- Semantik:
  - Ausführung Init-Anweisung
  - Auswertung Boolescher Ausdruck
  - Falls true:
    - Ausfürhung Anweisung bzw. Block
    - Berechnung Inkrement-Ausdruck
    - Erneute Auswertung Boolescher Ausdruck
    - usw.
  - Falls false: Beende `for`-Anweisung

for-Schleife (II)

- Beispiel:

```
(1) int i;
(2) for (i = 0; i <= 20; i++){
(3)   System.out.println(i);
(4) }
```

- Einsatz der for-Schleife:
  - Durchlaufen bekannten, zusammenhängendem Wertebereich
- Anmerkung:
  - Zählvariable kann in Initialisierungsanweisung definiert werden, d.h.
    `for(int i = 0; i <= 20; i++)`
    möglich
  - Gültigkeitsbereich der Variable `i`: `for`-Schleife

for-Schleife (III)

- Vereinfachte/verkürzte Syntax: `for (<Variable> : <Menge>) <Block>`
- Semantik:
  - Iteration über alle Elemente der Menge (z.B. eines Feldes), Bearbeitung im Block
- Beispiel:

```
(1) // Defintion eines Feldes
(2) int[] intFeld = {4,6,8,7,5,3,1};
(3) // "Herkömmliches" Ausgeben des Feldinhalts
(4) for (int i = 0; i < intFeld.length; i++)
(5)    {System.out.print(intFeld[i] + " ");}
(6) System.out.println();
(7) // Ausgeben Feldinhalt mit vereinfachter for-Schleife
(8) for (int j : intFeld){System.out.print(j + " ");}
```

führt zur Ausgabe

```
4 6 8 7 5 3 1
4 6 8 7 5 3 1
```

Übersicht

- Elementare Kontrollstrukturen
- Bedingte Anweisung
- Schleifen
- Sprünge
- Unterprogramme

Labels

- Syntax:
  `<Label-Anweisung> ::= <Label> ":" <Anweisung>`
  `<Label> ::= <Bezeichner>`
- Semantik: Keinen Einfluss auf Ablauf
- Beispiel:

```
(1) deklariere: int i = 0;
(2)             int grenze = 20;
(3)             do {
(4)                 ausgabe: System.out.println(i);
(5)                 i++;
(6)                }
(7)             while (i <= grenze);
```

break-Anweisung ohne Label

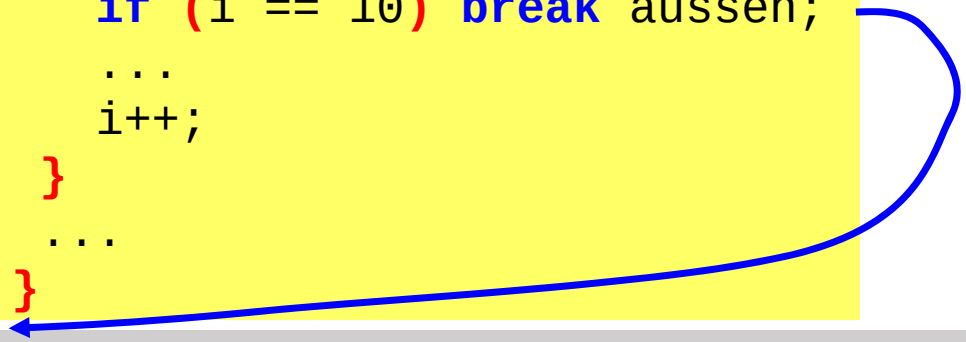
- Syntax:
  `<break-Anweisung> ::= "break;"`
- Semantik:
  - Verlassen innerste `do`-, `while`-, `for`- oder `switch`-Anweisung
- Beispiel:

```
( 1) int i = 0;
( 2) while(true){
( 3)    ...
( 4)    while(true){
( 5)       ...
( 6)       if (i == 10) break;
( 7)       ...
( 8)       i++;
( 9)    }
(10)    ...
(11) }
```

break-Anweisung mit Label

- Erweiterung Syntax break-Anweisung um Angabe eines Label: `break <Label>;`
- Semantik:
  - Verlassen umgebende `do`-, `while`-, `for`- oder `switch`-Anweisung mit Label, d.h. Sprung an das Ende dieser Anweisung
- Beispiel:

```
( 1)          int i = 0;
( 2) aussen: while(true){
( 3)            ...
( 4) innen:      while(true){
( 5)               ...
( 6)               if (i == 10) break aussen;
( 7)               ...
( 8)               i++;
( 9)             }
(10)             ...
(11)           }
```



switch-Anweisung (I)

- Größere Fallunterscheidung (geschachtelte `if`-Anweisung)
- Syntax:

```
<switch-Anweisung> ::= "switch" "(" <Ausdruck> ")"
                       "{" { <Fallunterscheidung> } "}"
<Fallunterscheidung>  ::=   <Anweisung>
                          | "case" <const-Ausdruck> ":„
                          | "default" ":"
```

- Bedingungen:
  - `const`-Ausdruck muss vom Typ `char`, `byte`, `short` oder `int` sein (Literal)
  - Ab Java 7 sind auch Strings zulässig (müssen in Anführungszeichen stehen)
  - Alle `const`-Ausdrücke in einer `switch`-Anweisung müssen vom selben Typ sein
  - Kein `const`-Ausdruck darf doppelt vorkommen
  - Typ von Ausdruck konform zum Typ der `const`-Ausdrücke
  - Höchstens ein `default`

switch-Anweisung (II)

- Semantik:
  - Auswertung Ausdruck
  - Existiert passender `const`-Ausdruck ⇒ Sprung an diese Stelle, Fortführung Programmausführung
  - Existiert kein passender `const`-Ausdruck, dann
    - Falls `default`-Ausdruck existiert, Fortsetzung an dieser Stelle
    - Sonst: Beende `switch`-Anweisung
- Beispiel:

```
(1) int i = 3;
(2) switch (i){
(3)   case 1  : System.out.println("Wert ist 1.");
(4)   case 2  : System.out.println("Wert ist 2.");
(5)   case 3  : System.out.println("Wert ist 3.");
(6)   case 4  : System.out.println("Wert ist 4.");
(7)   default : System.out.println(
(8)                              "Wert ist größer als 4.");}
```

switch-Anweisung (III)

führt zur Ausgabe

```
Wert ist 3.
Wert ist 4.
Wert ist größer als 4.
```

- Grund: Fortsetzung Ausführung am passenden `case`-Zweig
- Meistens nur dieser `case`-Zweig relevant ⇒ `break`-Anweisung einzubauen, d.h.

```
(1) int i = 3;
(2) switch (i){
(3)   case 1  : System.out.println("Wert ist 1."); break;
(4)   case 2  : System.out.println("Wert ist 2."); break;
(5)   case 3  : System.out.println("Wert ist 3."); break;
(6)   case 4  : System.out.println("Wert ist 4."); break;
(7)   default : System.out.println(
(8)                              "Wert ist größer als 4.");}
```

führt zur (erwarteten) Ausgabe

```
Wert ist 3.
```

switch-Anweisung (V)

- Anforderung: Gleiche Aktion für mehrere Ausdrücke
- Zwei Möglichkeiten:
  - Wiederholtes case
  - Ausdrücke durch Komma trennen (seit Java 12)
- Beispiel:

```
(1) int i = 3;
(2) switch (i){
(3)   case 1: case 2 :
(4)      System.out.println("Wert ist 1 oder 2."); break;
(5)   case 3: case 4 :
(6)      System.out.println("Wert ist 3 oder 4."); break;
(7)   default : System.out.println(
(8)                          "Wert ist größer als 4.");}
```

führt zur Ausgabe

```
Wert ist 3 oder 4.
```

switch-Anweisung (VI)

- Beispiel:

```
(1) int i = 3;
(2) switch (i){
(3)   case 1, 2 :
(4)      System.out.println("Wert ist 1 oder 2."); break;
(5)   case 3, 4 :
(6)      System.out.println("Wert ist 3 oder 4."); break;
(7)   default : System.out.println(
(8)                          "Wert ist größer als 4.");}
```

führt zur Ausgabe

```
Wert ist 3 oder 4.
```

switch-Anweisung (VI): Ausdrücke

- Anforderung: Mit switch-Anweisung Wert an Variable zurückgeben
- switch-Ausdrücke (seit Java 14) :
  - Verwendung Schlüsselwort yield
  - Entfallen break-Anweisung
- Beispiel:

```
(1) String s =
(2) switch (i){
(3)   case 1, 2: yield "Wert ist 1 oder 2.";
(4)   case 3, 4: yield "Wert ist 3 oder 4.";
(5)   default : yield "Wert ist größer als 4.";
(6) };
(7) System.out.println(s);
```

führt zur Ausgabe

```
Wert ist 3 oder 4.
```

switch-Anweisung (VII): Pfeilnotation

- Verwendung Pfeil statt Doppelpunkt (seit Java 14):
  - Vor dem Pfeil mehrere durch Komma getrennte Ausdrücke
  - Nach dem Pfeil einzelnes Statement oder Block
  - Wegfall `break`-Anweisung
- Beispiel:

```
(1) int i = 3;
(2) switch (i){
(3)   case 1, 2 -> System.out.println("Wert ist 1 oder 2.");
(4)   case 3, 4 -> {
(5)      String s = "Wert ist 3 oder 4.";
(6)      System.out.println(s);}
(7)   }
(8)   default : System.out.println("Wert ist größer als 4.");}
```

führt zur Ausgabe

```
Wert ist 3 oder 4.
```

switch-Anweisung (VIII): Pfeilnotation mit Ausdrücken

- Kombination von Pfeilnotation und switch-Ausdrücken (seit Java 14), Rückgabewert
  - Direkt hinter den Pfeil schreiben
  - Als Ausdruck oder Methodenaufruf hinter den Pfeil schreiben
  - Mit `yield`-Anweisung aus Block zurückgeben
- Beispiel:

```
(1) String s =
(2) switch (i){
(3)   case 1, 2 -> "Wert ist 1 oder 2.";
(4)   case 3, 4 -> "Wert ist " + "3 oder 4.";
(5)   case 5 -> { String t = "Wert ist 5."; yield t;}
(5)   default : yield "Wert ist größer als 5.";
(6) };
(7) System.out.println(s);
```

führt zur Ausgabe

```
Wert ist 3 oder 4.
```

Anmerkungen

- `yield` sog. kontextuelles Schlüsselwort (contextual keyword):
  - Nur innerhalb der `switch`-Anweisung verwendbar
  - Hintergrund:
    - Alte Programme könnten Variablen mit Namen yield haben
    - Diese bleiben damit korrekt
- Möglich (aber nicht empfehlenswert!):

```
(1) int i = 1;
(2) int s =
(3) switch (i){
(4)   case 1, 2 -> {int yield = 5; yield yield + yield;}
(5)   case 3, 4 -> yield 3;
(5)   default : 5;
(6) };
(7) System.out.println(s);
```

führt zur Ausgabe `10`

continue-Anweisung (I)

- Ermöglicht Sprung an das Ende einer Schleife
- Syntax: `<continue-Anweisung> ::= "continue" [ <Label> ] ";"`
- Semantik:
  - Kein Label angegeben ⇒ Sprung an das Ende der Schleife (und nächster Schleifendurchlauf)
  - Label angegeben und umgebende Schleife hat dieses Label ⇒ Sprung an das Ende dieser Schleife (und hier nächster Schleifendurchlauf)
- Beispiele:

```
(1) int erg = 0;
(2) for (int i=0;i<=10;i++){
(3)   if (i%2 == 0) continue; //Gerades i ⇒ Schleifenende
(4)   erg = erg + i;
(5) }
(6) System.out.println(erg);
```

continue-Anweisung (II)

- Beispiel mit innerer/äußerer Schleife:

```
( 1) int i=0,j=0,erg=0;
( 2) aussen:
( 3) while (i < 10){
( 4)   i++;
( 5)   innen:
( 6)   while (j < 10){
( 7)      j++;
( 8)      if (j%2 == 0) continue aussen;
( 9)      erg++;
(10)   }
(11) }
(12) System.out.println(erg);
```

- Ergebnis: 5
- Man sieht: `break`- und `continue`-Anweisungen (insb. mit Labeln) schlecht lesbar
- Daher: sparsam anwenden oder vermeiden

return-Anweisung

- Ermöglicht Verlassen einer Funktion und Zurückgeben eines Wertes
- Syntax: `<return-Anweisung> ::= "return" [ <Ausdruck> ] ";"`
- Semantik:
  - Aktuelle Funktion/Prozedur/Methode wird unmittelbar verlassen
  - Ist ein Ausdruck angegeben, wird dieser als Funktionswert zurückgegeben
- Beispiel:

```
(1) int erg = 0;
(2) for (int i=0;i<=10;i++){
(3)   if (i%2 == 0) continue; //Gerades i ⇒ Schleifenende
(4)   erg = erg + i;
(5) }
(6) return erg;
```

- Anm.: Behandlung Methoden später ausführlich

Unbedingte Sprünge

- In manchen (älteren) Sprachen gibt es unbedingte Sprünge
- Syntax: `goto <Label>`
- Verwendung führt schnell zu schlecht lesbarem Programmcode
- Daher:
  - Nur sparsam bzw. möglichst nicht verwenden
  - In modernen Sprachen (z.B. Java) nicht vorhanden
- „Go To Statement Considered Harmful“ (Artikel von Dijkstra 1968)

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code).

dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** *B* **repeat** *A* or **repeat** *A* **until** *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of

Übersicht

- Elementare Kontrollstrukturen
- Bedingte Anweisung
- Schleifen
- Sprünge
- Unterprogramme

Definition (I)

- Zusammenfassung einer Menge von Anweisungen unter einem Namen
- Syntax:

```
<Unterprogramm> ::=
        <Rückgabetyp> <Unterprogramm-Name>
        "(" { <Parametertyp> <Parametername> ,}
           |  <Parametertyp> <Parametername>
        "){" <Anweisungen des Unterprogramms> "}"
```

- Benennungen:
  - Unterprogrammen heißen auch Subroutinen oder Co-Routinen (veraltet)
  - Prozedurale Welt:
    - Unterprogramme mit Rückgabetyp heißen Funktionen
    - Unterprogramme ohne Rückgabe heißen Prozeduren (Verwendung von `void` als Rückgabetyp)
  - In OO-Sprachen heißen alle Unterprogramme Methoden
  - Anweisungen des Unterprogramms werden auch als Rumpf bezeichnet

Definition (II)

- Semantik:
  - Unterprogramm wird aus anderem Unterprogramm (oder aus dem Hauptprogramm (in Java `main`-Methode)) aufgerufen
  - Binden aktueller Parameter an formale Parameter
  - Abarbeiten des Unterprogramms
  - Funktionen: Bei `return`-Anweisung Rückgabe Ausdruck und Verlassen Unterprogramm
  - Prozeduren: Abarbeitung bis zum Ende des Rumpfes

Beispiel (I): Quadrat berechnen

```
( 1) class Quadrat{
( 2)   static int quadrat(int i){
( 3)     return i * i;
( 4)   }
( 5)
( 6)   public static void main (String[] args){
( 7)      int j, k = 7;
( 8)      j = quadrat(k);
( 9)      System.out.println(j);
(10)      j = quadrat(9);
(11)      System.out.println(j);
(12)      System.out.println(quadrat(11));
(13)   }
(14) }
```

Kopf

Rumpf

Unterprogramm (Funktion, Methode)

Rückgabetyp

Name

Parameterliste

Beim Aufruf:
- Parameterübergabe (hier 9 an i)
- Abarbeiten Anweisungen im Rumpf
- Ausdruck hinter `return` zurückgeben

führt zu Ausgabe

```
49
81
121
```

Beispiel (II): Maximum

```
( 1) class Maximum{
( 2)   static int maximum(int i, int j, int k){
( 3)     int m = i;
( 4)     if (j>m){m = j;}
( 5)     if (k>m){m = k;}
( 6)     return m;
( 7)   }
( 8)   public static void main (String[] args){
( 9)      int a = 3, b = 1, c = 7;
(10)      System.out.println(maximum(a,b,c));
(11)      System.out.println(maximum(5,3,1));
(12)      System.out.println(maximum(11,b,c));
(13)   }
(14) }
```

führt zu Ausgabe

```
7
5
11
```

Zusammenfassung (I)

- Elementare Kontrollstrukturen:
  - Deklaration
  - Ausdruck
  - Zuweisung
  - Block
  - Gültigkeitsbereich und Lebensdauer
  - Sequenz
  - Leere Anweisung

- Bedingte Anweisung:
  - `if`-Anweisung
  - `else`-Zweig
  - `elseif`

Zusammenfassung (II)

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

- Sprünge:
  - Label
  - Unterbrechung (`break`)
  - `switch`-Anweisung
  - Pfeilnotation
  - `switch`-Ausdrücke
  - Fortsetzung (`continue`)
  - Rückgabe (`return`)
  - Unbedingte Sprünge

Zusammenfassung (III)

- Unterprogramme:
  - Funktionen
  - Prozeduren
  - Methoden
  - Kopf, Rumpf
  - Rückgabetyp
  - Parameterliste
  - Parameterübergabe

Aufgaben (I)

- **Aufgabe 1**

  Deklarieren Sie eine Integer-Konstante `MAXIMUM`. Schreiben Sie ein Programm, das zunächst alle geraden Zahlen zwischen `1` und `MAXIMUM` in aufsteigender Reihenfolge und danach alle ungeraden Zahlen zwischen `1` und `MAXIMUM` in absteigender Reihenfolge ausgibt.

- **Aufgabe 2**

  Schreiben Sie ein Programm, das für alle ganzen Zahlen n mit $1 \leq n \leq 20$ den Wert $\sum_{i=1}^{n} i$ (d.h. die Summe aller Zahlen von *1* bis *n*) berechnet und ausgibt (also die Folge 1, 3, 6, 10, 15 ...).

  Realisieren Sie dies mit drei verschiedenen Wiederholungsanweisungen!

Aufgaben (II)

- **Aufgabe 3**

  Das Sieb des Erathostenes (nach Eratosthenes von Cyrene ca. 276 - 194 v.u.Z.) ist ein Algorithmus zur Bestimmung aller Primzahlen zwischen 1 und einer festgelegten oberen Schranke.

  Erathostenes ging dabei (damals noch ohne Computer) wie folgt vor:

  1. Er schrieb alle Zahlen von 1 bis s in einer Reihe auf.
     Beispielsweise mit s = 15: 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
  2. Er nahm zuerst die 2 als Basis und strich alle ihre Vielfachen durch:
     1,2,3,~~4~~,5,~~6~~,7,~~8~~,9,~~10~~,11,~~12~~,13,~~14~~,15
  3. Dann nahm er die nächste noch nicht durchgestrichene Zahl (also zunächst die 3) als Basis und strich alle ihre Vielfachen durch:
     1,2,3,~~4~~,5,~~6~~,7,~~8~~,~~9~~,~~10~~,11,~~12~~,13,~~14~~,~~15~~

Aufgaben (III)

4. Er überprüfte, ob die letzte Zahl, die er als Basis gewählt hatte, größer als s/2 war.
   Falls nein, wiederholte er Schritt 3 mit der nächsten noch nicht durchgestrichenen Zahl.
   Falls ja, war er fertig.
   Die noch nicht durchgestrichenen Zahlen entsprechen genau der Menge der Primzahlen zwischen 1 und s.
   Entwickeln Sie ein Programm, das das Sieb des Erathostenes anwendet, um alle Primzahlen zwischen 1 und einer Integer-Konstante `MAXIMUM` auszugeben!

- **Aufgabe 4**

  Gegeben seien die Variablen Startwert, Laufzeit und drei verschiedene Zinssätze.
  Schreiben Sie ein Programm, das nebeneinander die Entwicklung der Verzinsung über die Jahre angibt!

Aufgaben (IV)

- **Aufgabe 5**

  Der Goldfrosch ist ein virtuelles Tier, das sich auf der Zahlengerade mit einer Schrittweite von +1 oder –2 bewegen kann. An jeder Stelle i = 1, 2, 3, ... ist ein Richtungspfeil angebracht, der nach rechts oder links weist. Landet ein Frosch auf Platz i, so dreht er den Richtungspfeil zunächst um und hüpft dann in der neuen Richtung, wobei folgende Regeln gelten: Zeigt der Pfeil nach links, hüpft der Frosch von Platz i nach i-2, andernfalls von Platz i nach Platz i+1. Auf Platz 0 und Platz –1 steht jeweils ein Eimer, in den der Frosch plumpst, wenn er zu weit nach links abgekommen ist. In diesem Fall startet ein neuer Frosch auf Platz 1. Zu Beginn weisen alle Richtungspfeile nach rechts.

  a) Schreiben Sie ein Java-Programm (unter Verwendung der bisher bekannten Sprachmittel), das den Froschprozess nachvollzieht und am Ende die Anzahl der Frösche in den beiden Eimern angibt!
  b) Wie ist das Verhältnis der Frosch-Anzahlen in beiden Eimern auf lange Sicht?
  c) Kann es vorkommen, dass ein Frosch nach rechts ins Unendliche wegspringt?

Aufgaben (V)

- **Aufgabe 6**

  Schreiben Sie ein Java-Programm, in dem Sie zunächst eine Enumeration „Monate“ mit den Monatsnamen anlegen.
  Verwenden Sie anschließend eine `switch`-Anweisung, die den Monaten Quartale zuordnet (Zuweisung an String-Variable `quartal`). Formulieren Sie diese auf verschiedene Art und Weise:

  a) „ursprüngliche“ `switch`-Anweisung
  b) Zusammenfassung mehrerer Ausdrücke
  c) Pfeilnotation
  d) Zusammenfassung mehrerer Ausdrücke in Pfeilnotation
  e) `switch`-Ausdruck mit Pfeilnotation