Kapitel 7: Bedingte Anweisungen und Iteration i

Verzweigung (bedingte Anweisung)

- Analog zum bedingten Ausdruck führen wir nun noch die *bedingte Anweisung* ein.
- Damit lässt sich der Fluss von Anweisung steuern: die bedingte Anweisung ermöglicht *Verzweigung*.
- Durch die bedingten Anweisungen kann man Prozeduren auch rekursiv formulieren (analog wie bei Funktionen).

Verzweigung (bedingte Anweisung)

- Die Syntax von einer Bedingten Anweisung lautet in unserer Pseudo-Schreibweise:

  IF b **THEN** $\mathfrak{a}_1$ **ELSE** $\mathfrak{a}_2$ **ENDIF**

- Dabei ist b ein Ausdruck vom Typ $\mathbb{B}$, $\mathfrak{a}_1$ und $\mathfrak{a}_2$ sind beliebige Anweisungsfolgen.
- Der Teil **ELSE** $\mathfrak{a}_2$ kann auch fehlen.
- Eine bedingte Anweisung kann überall dort stehen, wo eine beliebige Anweisung stehen darf.
- Damit kann man bedingte Anweisungen auch ineinander schachteln.

Verzweigung (bedingte Anweisung)

Eine bedingte Anweisung $\mathfrak{a}$ ändert einen Zustand $\mathcal{S}$ wie folgt:

- Einfache Verzweigung (**ELSE** $\mathfrak{a}_2$ fehlt):
  Ist $\mathfrak{a}$ der Form **IF** *b* **THEN** $\mathfrak{a}_1$ **ENDIF** , wobei $\mathfrak{a}_1$ den Zustand $\mathcal{S}$ in einen Zustand $\mathcal{S}_1$ überführt, d.h. $\mathcal{S} \xrightarrow{\mathfrak{a}_1} \mathcal{S}_1$
  Dann: Wenn der Ausdruck *b* im Zustand $\mathcal{S}$ den Wert $W_{\mathcal{S}}(b) = \mathit{TRUE}$ erhält, führt $\mathfrak{a}$ den Zustand $\mathcal{S}$ in den Nachfolgezustand $\mathcal{S}_1$ über, andernfalls ($W_{\mathcal{S}}(b) = \mathit{FALSE}$) bleibt der Zustand unverändert,

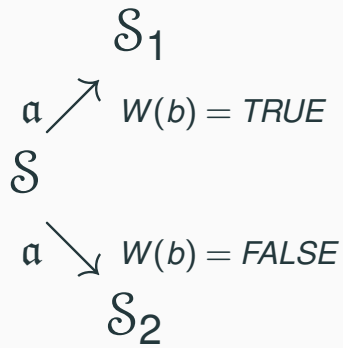
  d.h. es gilt:

$$\begin{array}{ccc} & & \mathcal{S}_1 \\ & \mathfrak{a} \nearrow & W(b) = \mathit{TRUE} \\ \mathcal{S} & & \\ & \mathfrak{a} \searrow & W(b) = \mathit{FALSE} \\ & & \mathcal{S} \end{array}$$

Verzweigung (bedingte Anweisung)

- Zweifache Verzweigung:
  Ist $\mathfrak{a}$ der Form **IF** *b* **THEN** $\mathfrak{a}_1$ ESLE $\mathfrak{a}_2$ **ENDIF** , wobei $\mathfrak{a}_1$ den Zustand $\mathcal{S}$ in einen Zustand $\mathcal{S}_1$ und $\mathfrak{a}_2$ den Zustand $\mathcal{S}$ in einen Zustand $\mathcal{S}_2$ überführt, d.h. $\mathcal{S} \xrightarrow{\mathfrak{a}_1} \mathcal{S}_1$ bzw. $\mathcal{S} \xrightarrow{\mathfrak{a}_2} \mathcal{S}_2$
  Dann: Wenn der Ausdruck *b* im Zustand $\mathcal{S}$ den Wert $W_{\mathcal{S}}(b) = \mathit{TRUE}$ erhält, führt $\mathfrak{a}$ den Zustand $\mathcal{S}$ in den Nachfolgezustand $\mathcal{S}_1$ über, andernfalls ($W_{\mathcal{S}}(b) = \mathit{FALSE}$) in den Nachfolgezustand $\mathcal{S}_2$,

  d.h. es gilt:



Beispiel

```
PROCEDURE werIstGroesser(x : ℕ0, y : ℕ0) → ℕ
  OUTPUT  die Stelle mit der groesseren Zahl
  BODY
    VAR  erg : ℕ;
    IF  x > y  THEN  erg ← 1;  ELSE  erg ← 2;  ENDIF
    RETURN  erg;
```

Auswertung von $werIstGroesser(3,1)$:

$\mathcal{S}_0$: $\{(x,3),(y,1)\}$

$\downarrow$ **VAR** $erg : \mathbb{N}$;

$\mathcal{S}_1$: $\{(x,3),(y,1),(erg,\omega)\}$

$\downarrow$ **IF** $x > y$ ...

$W(x > y) = 3 > 1 = TRUE$

d.h. $erg \leftarrow 1$; wird ausgeführt

$\mathcal{S}_2$: $\{(x,3),(y,1),(erg,1)\}$

RETURN erg; gibt 1 zurück.

Auswertung von $werIstGroesser(1,3)$:

$\mathcal{S}_0$: $\{(x,1),(y,3)\}$

$\downarrow$ **VAR** $erg : \mathbb{N}$;

$\mathcal{S}_1$: $\{(x,1),(y,3),(erg,\omega)\}$

$\downarrow$ **IF** $x > y$ ...

$W(x > y) = 1 > 3 = FALSE$

d.h. $erg \leftarrow 2$; wird ausgeführt

$\mathcal{S}_2$: $\{(x,3),(y,1),(erg,2)\}$

RETURN erg; gibt 2 zurück.

Verzweigung (bedingte Anweisung)

- Wie erwähnt ist nun auch Rekursion im imperativen Paradigma möglich:

PROCEDURE $fibI(n : \mathbb{N}_0) \to \mathbb{N}_0$
  OUTPUT die n-te Fibonacci-Zahl
  BODY
    VAR $erg : \mathbb{N}$;
    IF $n = 0 \vee n = 1$ **THEN** $erg \leftarrow 1$;
      ELSE $erg \leftarrow fibI(n-1) + fibI(x-2)$; **ENDIF**
    RETURN erg;

- Eine Auswertung der Zustandsübergänge für einen Beispielaufruf (etwa $fibI(3)$) sei dem fleißigen Leser überlassen.

Verzweigung (bedingte Anweisung)

Ein etwas komplexeres Beispiel:

- Wir wollen mit vier ja/nein-Fragen den Geburtsmonat einer Person herausfinden.
- Die Fragen sind z.B. (Beispielantworten für April):
  1. Haben Sie im ersten Halbjahr Geburtstag? (ja)
  2. Haben Sie im ersten oder dritten (also in einem ungeraden) Quartal Geburtstag? (nein)
  3. Haben Sie in einem geraden Monat Geburtstag? (ja)
  4. Ist es der erste Monat im Quartal? (ja)
- Wir modellieren die Antwort auf diese vier Fragen mit vier Variablen aus $\mathbb{B}$ (Antwort ja = *TRUE*): *eHJ*, *uQ*, *gM*, *eM*.

Verzweigung (bedingte Anweisung)

```
PROCEDURE gebMonat(eHJ : 𝔹, uQ : 𝔹, gM : 𝔹, eM : 𝔹) → ℕ
  OUTPUT  der Geburtsmonat
  BODY
  VAR  erg : ℕ;
  IF  eHJ  THEN  // 1.HJ
    IF  uQ  THEN  // Q1
      IF  gM  THEN  erg ← 2;
      ELSE
        IF  eM  THEN  erg ← 1;
        ELSE  erg ← 3;
        ENDIF
      ENDIF
    ELSE  // Q2
      IF  gM  THEN
        IF  eM  THEN  erg ← 4;
        ELSE  erg ← 6;
        ENDIF
      ELSE  erg ← 5;
      ENDIF
    ENDIF
  ELSE
    ...  (Zweites Halbjahr)
  ENDIF

  RETURN  erg;
```

1. Halbjahr		
	Jan	1
Q1	Feb	2
	Mar	3
	Apr	4
Q2	Mai	5
	Jun	6
2. Halbjahr		
	Jul	7
Q3	Aug	8
	Sep	9
	Okt	10
Q4	Nov	11
	Dez	12

Iteration

- Nun fehlt uns zur Steuerung der Anweisungsfolge nur noch die *Wiederholungsanweisung*, um *Iteration* umzusetzen.
- Es gibt grundsätzlich verschiedene Arten, wie Iteration umgesetzt werden kann.
- Manche Programmiersprachen bieten verschiedene Varianten und auch Mischformen an.
- Tatsächlich sind diese Varianten i.d.R. äquivalent[12], daher beschränken wir uns hier auf die *bedingte Wiederholungsanweisung* (*Schleife*).
- Eine populäre Variante ist die *gezählte Wiederholungsanweisung* (*Schleife*).

[12] Das werden wir später in den Übungen mit Iteration in Java sehen

Iteration

- Auch die bedingte Wiederholung gibt es in mehreren (äquivalenten) Varianten, von denen wir die vielleicht bekannteste verwenden:
- Eine bedingte Wiederholungsanweisung hat die Form:

  `WHILE` b **`DO`** $\mathfrak{a}_1$ **`ENDDO`**

- Dabei ist b ein Ausdruck vom Typ $\mathbb{B}$ und $\mathfrak{a}_1$ ist eine Anweisung bzw. ein Anweisungsblock.
- Eine bedingte Wiederholungsanweisung kann überall dort stehen, wo eine beliebige Anweisung stehen darf.
- Damit kann man bedingte Anweisungen auch wieder ineinander schachteln.

- Formalisierung der Zustandsänderung:
  - Sei $\mathfrak{a}$ der Form **WHILE** b **DO** $\mathfrak{a}_1$ **ENDDO**.
  - Die Anweisung(sfolge) $\mathfrak{a}_1$ bewirkt eine definierte Zustandsänderung $\Delta(\mathcal{S})$ von Zustand $\mathcal{S}$ bewirkt.

  $\mathfrak{a}$ überführt den Zustand $\mathcal{S}$ in einen Nachfolgezustand $\hat{\mathcal{S}}$ wie folgt:
  - Ist $W_{\mathcal{S}}(b) = \mathit{TRUE}$, so wird $\mathcal{S}$ in $\Delta(\mathcal{S})$ überführt, ansonsten bleibt $\mathcal{S}$ erhalten.
  - Ist $W_{\Delta(\mathcal{S})}(b) = \mathit{FALSE}$, wird $\Delta(\mathcal{S})$ in $\Delta(\Delta(\mathcal{S}))$ überführt.
  - Dies wird solange iteriert, bis ein Zustand $\hat{\mathcal{S}} = \Delta(\ldots\Delta(\mathcal{S})\ldots)$ erreicht ist, in dem erstmals $W_{\hat{\mathcal{S}}}\, b = \mathit{FALSE}$ gilt.
  - Dann gilt: $\mathcal{S} \xrightarrow{\mathfrak{a}} \hat{\mathcal{S}}$ bzw.
    $\mathcal{S} \xrightarrow{\mathfrak{a}_1} \Delta(\mathcal{S}) \xrightarrow{\mathfrak{a}_1} \Delta(\Delta(\mathcal{S})) \xrightarrow{\mathfrak{a}_1} \ldots \xrightarrow{\mathfrak{a}_1} \Delta(\ldots\Delta(\mathcal{S})\ldots) = \hat{\mathcal{S}}$

- Beispiel: der Euklidische GGT Algorithmus:

```
PROCEDURE ggt(x : ℕ, y : ℕ) → ℕ
  OUTPUT  der GGT von x und y
  BODY
  WHILE  x ≠ y  DO
    IF  x > y  THEN  x ← x − y;  ELSE  y ← y − x; ENDIF
  ENDDO
  RETURN  x;
```

- Auswertung von $ggt(12,44)$:
  - Ausgangssituation: $\mathcal{S}_0 = \{(x,12),(y,44)\}$
  - Zustandsübergänge von **WHILE** $x \neq y$ **DO**...:
  - $W_{\mathcal{S}_0}(x \neq y)$ ist *TRUE*, daher wird **IF**...**THEN**...**ELSE**... ausgeführt:
    Der Wert des Wächters ist $W_{\mathcal{S}_0}(x > y)$ ist *FALSE*, daher wird $y \leftarrow y - x$; ausgeführt und folgender Zustand erreicht:
    $\Delta(\mathcal{S}_0) = \{(x,12),(y,44-12)\} = \{(x,12),(y,32)\}$

- Auswertung von $ggt(12,44)$ (cont.):

```
PROCEDURE ggt(x : ℕ, y : ℕ) → ℕ
  OUTPUT  der GGT von x und y
  BODY
  WHILE  x ≠ y  DO
    IF  x > y  THEN  x ← x − y;  ELSE  y ← y − x; ENDIF
  ENDDO
  RETURN  x;
```

  - $W_{\Delta(\mathcal{S}_0)}(x \neq y)$ ist immer noch *TRUE*:
    Der Wert des Wächters ist $W_{\Delta(\mathcal{S}_0)}(x > y) = W(12 > 32)$ ist *FALSE*, daher wird $y \leftarrow y - x;$ ausgeführt und folgender Zustand erreicht:
    $\Delta(\Delta(\mathcal{S}_0)) = \{(x,12),(y,32-12)\} = \{(x,12),(y,20)\}$
  - $W_{\Delta(\Delta(\mathcal{S}_0))}(x \neq y)$ ist immer noch *TRUE*:
    Der Wert des Wächters ist $W_{\Delta(\mathcal{S}_0)}(x > y) = W(12 > 20)$ ist *FALSE*, daher wird $y \leftarrow y - x;$ ausgeführt und folgender Zustand erreicht:
    $\Delta(\Delta(\Delta(\mathcal{S}_0))) = \{(x,12),(y,20-12)\} = \{(x,12),(y,8)\}$

- Auswertung von $ggt(12,44)$ (cont.):

```
PROCEDURE ggt(x : ℕ, y : ℕ) → ℕ
  OUTPUT  der GGT von x und y
  BODY
  WHILE  x ≠ y  DO
    IF  x > y  THEN  x ← x − y;  ELSE  y ← y − x; ENDIF
  ENDDO
  RETURN  x;
```

  - $W_{\Delta(\Delta(\Delta(\mathcal{S}_0)))}(x \neq y)$ ist immer noch *TRUE*:
    Der Wert des Wächters ist $W_{\Delta(\Delta(\Delta(\mathcal{S}_0)))}(x > y) = W(12 > 8)$ ist *TRUE*, daher wird nun $x \leftarrow x - y;$ ausgeführt und folgender Zustand erreicht:
    $\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0)))) = \{(x, 12-8), (y, 8)\} = \{(x, 4), (y, 8)\}$
  - $W_{\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0))))}(x \neq y)$ ist immer noch *TRUE*:
    Der Wert des Wächters ist $W_{\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0))))}(x > y) = W(4 > 8)$ ist *FALSE*, daher wird wieder $y \leftarrow y - x;$ ausgeführt und folgender Zustand erreicht:
    $\Delta(\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0))))) = \{(x, 4), (y, 8-4)\} = \{(x, 4), (y, 4)\}$

- Auswertung von $ggt(12,44)$ (cont.):

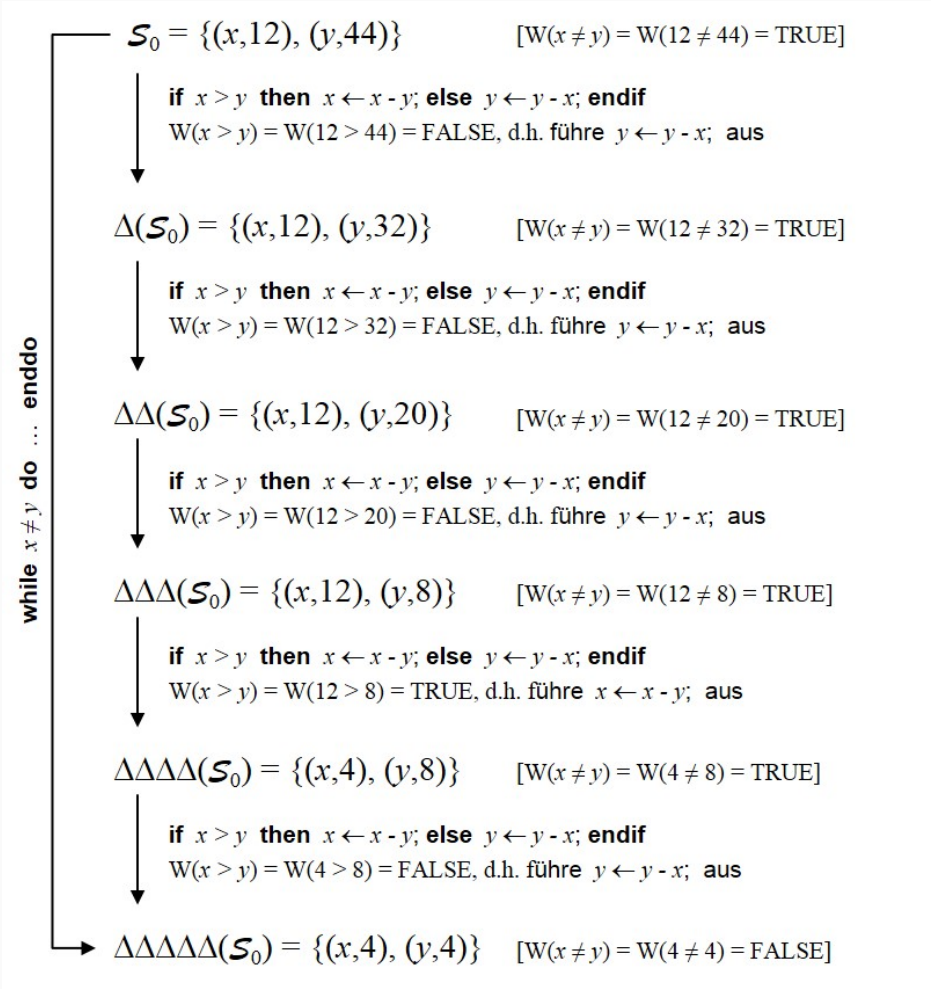
```
    PROCEDURE ggt(x : ℕ, y : ℕ) → ℕ
      OUTPUT  der GGT von x und y
      BODY
(*)   WHILE  x ≠ y  DO
        IF  x > y  THEN  x ← x − y;  ELSE  y ← y − x; ENDIF
      ENDDO
      RETURN  x;
```

  - $W_{\Delta(\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0)))))}(x \neq y)$ ist nun *FALSE* $(4 \neq 4)$, d.h. $\Delta(\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0))))) = \{(x,4),(y,4)\}$ ist der Nachfolgezustand von $\mathcal{S}_0$ bzgl. der bedingten Anweisung in Zeile (*).
  - **RETURN** x; gibt somit $W_{\Delta(\Delta(\Delta(\Delta(\Delta(\mathcal{S}_0)))))}(x) = 4$ zurück.

Iteration

- Und hier nochmal „schön“

```
PROCEDURE ggt(x : ℕ, y : ℕ) → ℕ
    OUTPUT   der GGT von x und y
    BODY
    WHILE   x ≠ y   DO
       IF x > y THEN x ← x − y;
       ELSE y ← y − x; ENDIF
    ENDDO
    RETURN   x;
```



Iteration und Rekursion

- Eine Schleife kann Rekursion „simulieren“
- Hier eine sehr ähnliche, aber rekursive (und daher funktionale) Variante von *ggt* (mit zwei verschachtelten bedingten Ausdrücken):

FUNCTION ggtRek($x : \mathbb{N}, y : \mathbb{N}$) $\rightarrow \mathbb{N}$
  OUPUT der GGT von x und y rekursiv
  BODY
  IF $x = y$ **THEN** x
  ELSE
    IF x>y **then** ggtRek($x - y, y$)
    ELSE ggtRek($x, y - x$)
    ENDIF
  ENDIF

Kapitel 8: Verzweigung/Iteration in Java i

Verzweigung (bedingte Anweisung) in Java

- Java erlaubt ebenfalls zwei Formen der bedingten Anweisung:
  - Eine *einfache bedingte Anweisung*:
    `if(<Bedingung>) <Anweisung>`
  - Eine *bedingte Verzweigung*:
    `if(<Bedingung>) <Anweisung1> else <Anweisung2>`

  wobei
  - `<Bedingung>` ein Ausdruck vom Typ **`boolean`** ist
  - `<Anweisung>`, `<Anweisung1>` und `<Anweisung2>` jeweils eine (ggfs. auch leere) Anweisung ist (natürlich ist auch ein Block mit mehreren Anweisungen eine Anweisung).

Verzweigung (bedingte Anweisung)

- Beispiel: der Algorithmus `ggTRek`, in Java:

```
public static int ggTRek(int x, int y) {
      if(x == y) { return x; }
      else {
            if(x > y) { return ggTRek(x-y,y); }
            else { return ggTRek(x,y-x); }
      }
}
```

Verzweigung (bedingte Anweisung)

- **Achtung**: nicht gesetzte Blockklammern bergen Fehlerpotential, z.B. das sog. Dangling Else:

```
if (b)
      if (b2)
            a2;
else
      a;
```

  - Zu welchem **`if`**-Statement gehört der **`else`**-Zweig?
  - Zur inneren Verzweigung **`if`** `(b2)`
  - (Die (falsche!) Einrückung ist belanglos für den Compiler und verführt den menschlichen Leser hier, das Programm falsch zu interpretieren)

Verzweigung (bedingte Anweisung)

- Mit Blockklammern wäre das nicht passiert:

```
if (b) {
     if (b2) { a2; }
     else { a; }
}
```

oder

```
if (b) {
     if (b2) { a2; }
}
else { a; }
```

Sprunganweisung

- In Java gibt es eine weitere Möglichkeit, spezielle Mehrfachverzweigungen (sog. *Sprunganweisungen*) auszudrücken.
- Die sog. **`switch`**-Anweisung funktioniert allerdings etwas anders als bedingte Anweisungen:

```
switch (<Ausdruck>) {
      case <Konstante1> : <Anweisungsfolge_1>
      case <Konstante2> : <Anweisungsfolge_2>
      ...
      default : <Anweisungsfolge_Default>
}
```

Sprunganweisung

- **Achtung**: der Ausdruck `<Ausdruck>` darf nur vom Typ **`byte, short, int`** oder **`char`** sein.
- Die Konstanten `<Konstante_i>` und der Ausdruck `<Ausdruck>` müssen den selben Typ haben.
- Abhängig vom Wert des Ausdrucks `<Ausdruck>` wird die Sprungmarke angesprungen, deren Konstante `<Konstante_i>` mit dem Wert von `<Ausdruck>` übereinstimmt.
- Die **`case`**-Marken sollten alle verschieden sein, müssen aber nicht.

Sprunganweisung

- Die Anweisungen nach der Sprungmarke, die als erstes mit dem Wert von `<Ausdruck>` übereinstimmt, werden ausgeführt (es sind keine Blockklammern nötig).
- *Achtung*: Es werden alle Anweisungen hinter dieser Marke ausgeführt. Es erfolgt *keine* Unterbrechung, wenn das nächste Label erreicht wird, sondern es wird dort fortgesetzt! Dies ist eine beliebte Fehlerquelle!
- Eine Unterbrechung kann durch die Anweisung **`break;`** erzwungen werden, die direkt zum Ende der **`switch`**-Anweisung verzweigt.

Sprunganweisung

- Die optionale **`default`**-Marke wird dann angesprungen, wenn keine passende Sprungmarke gefunden wird.
- Fehlt die **`default`**-Marke und wird keine passende Sprungmarke gefunden, so wird keine Anweisung innerhalb der **`switch`**-Anweisung ausgeführt.
- Nach einer Marken-Definition **`case`** kann i.Ü. auch die leere Anweisung stehen.

Sprunganweisung

- Beispiel

```
switch (month) {
  case 1: case 3: case 5: case 7: case 8: case 10: case 12:
    days = 31; break;
  case 4: case 6: case 9: case 11:
    days = 30; break;
  case 2:
    if (leapYear) {
      days = 29;
    } else {
      days = 28;
    }
    break;
  default :  /* leerer Zweig */
}
```

Iteration

- Java kennt mehrere Arten von *Iteration*, hier zunächst *bedingten Schleifen*:
  - Die klassische While-Schleife (so wie wir sie in der Theorie kennen gelernt haben):

    ```
    while (<Bedingung>) <Anweisung>
    ```
  - Die Do-While-Schleife:

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

  dabei bezeichnet `<Bedingung>` einen Ausdruck vom Typ **`boolean`** und `<Anweisung>` ist eine (ggfs. auch leere) Anweisung.

Iteration

- Unterschied: `<Anweisung>` wird einmal *vor* (Do-While) bzw. *nach* (While) der Überprüfung von `<Bedingung>` ausgeführt.
- Dannach wird wieder an den Beginn der Wiederholungsanweisung gesprungen.
- Hat `<Bedingung>` den Wert **`false`**, wird die Schleife verlassen, d.h. mit der Anweisung nach der Wiederholungsanweisung fortgefahren.
- Bis auf die erste Ausführung des Schleifenrumpfes sind also beide Varianten äquivalent und ihr Zustandsänderungen entsprechen unserer theoretischen Formalisierung.

- Beispiel: die Fakultätsfunktion nicht-rekursiv

Mit While-Schleife:

```
public static int fakWhile(int x) {
      int erg = 1;
      int laufvariable = 1;
      while(laufvariable<=x) {
            erg = erg * laufvariable;
            laufvariable++;
      }
      return erg;
}
```

Mit Do-While-Schleife:

```
public static int fakDoWhile(int x) {
      int erg = 1;
      int laufvariable = 1;
      do {
            erg = erg * laufvariable;
            laufvariable++;
      } while (laufvariable<x);
      return erg;
}
```

Weitere Alternative:

```
public static int fakWhile02(int x) {
      int erg = 1;
      while(x>0) {
            erg = erg * x;
            x--;
      }
      return erg;
}
```

Iteration

- Ein weiteres Beispiel:

```
public static int ggt(int x, int y) {
      while(x != y) {
            if(x>y) {
                  x = x - y;
            } else {
                  y = y - x;
            }
      return x;
}
```

Iteration

- Eine weitere bedingte Schleife kann in Java mit dem Schlüsselwort **`for`** definiert werden:

```
for (<Initialisierung>; <Bedingung>; <Update>)
    <Anweisung>
```

- Alle drei Bestandteile im Schleifenkopf sind Ausdrücke (`<Bedingung>` muss vom Typ **`boolean`** sein).
- Vorsicht: Dieses Konstrukt ist keine klassische *gezählte Schleife*[13], die es in anderen Programmiersprachen gibt.

[13] Hier wird der Schleifenrumpf entsprechend einer Zählers, der hoch bzw.runter gezählt wird, durchlaufen.

Iteration

- Der Ausdruck `<Initialisierung>`
  - wird einmal vor dem Start der Schleife aufgerufen
  - darf Variablendeklarationen mit Initialisierung enthalten (um einen Zähler zu erzeugen); diese Variable ist nur im Schleifenkopf und innerhalb der Schleife (`<Anweisungfolge>`) sichtbar aber nicht außerhalb
  - darf auch fehlen (u.a. in diesem Fall ist es eben keine Zählschleife)

Iteration

- Der Ausdruck `<Bedingung>`
  - ist ähnlich wie bei den While-Konstrukten die Testbedingung
  - wird zu Beginn jedes Schleifendurchlaufs überprüft
  - die Anweisung(en) `<Anweisung>` wird (werden) nur ausgeführt, wenn der Ausdruck `<Bedingung>` den Wert **`true`** hat
  - kann fehlen (dies ist dann gleichbedeutend mit dem Ausdruck **`true`**)
- Der Ausdruck `<Update>`
  - verändert üblicherweise den Schleifenzähler (falls vorhanden)
  - wird am Ende jedes Schleifendurchlaufs ausgewertet
  - kann fehlen

Iteration

- Eine *gezählte Schleife* wird in Java wie folgt mit Hilfe der **`for`**-Schleife notiert (hier mit aufsteigender Zählweise):

```
for (<Zaehler>=<Startwert>;
     <Zaehler> <= <Endwert>;
     <Zaehler> = <Zaehler> + <Schrittweite>)
     <Anweisung>
```

- Beispiel: Nochmal Fakultät

```
public static int fakultaetFor(int n) {
     int erg = 1;
     for(int i = 1; i <= n; i++) {
          erg = erg * i;
     }
     return erg;
}
```

Iteration

- In Java gibt es Möglichkeiten, die normale Auswertungsreihenfolge innerhalb einer **`do`**-, **`while`**- oder **`for`**-Schleife zu verändern.
- Der Befehl **`break`** beendet die Schleife sofort und das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Der Befehl **`continue`** beendet die aktuelle Iteration und beginnt mit der nächsten Iteration, d.h. es wird an den Beginn des Schleifenrumpfes gesprungen.
- Sind mehrere Schleifen ineinander geschachtelt, so gilt der **`break`** bzw. **`continue`**-Befehl nur für die aktuelle (innerste) Schleife.

Iteration

- Die Befehle **`break`** und **`continue`** können in Java auch für (eingeschränkte) *Sprungbefehl* benutzt werden, um an eine beliebige Stelle in einem Programm springen.
- Mit Sprungbefehlen wird ein Programm allerdings sehr schnell sehr unübersichtlich! Wir raten dringend von der Verwendung ab und präsentieren daher auch kein Beispiel.
- Die Verwendung von **`break`** und **`continue`** ist i.Ü. grundsätzlich schlechter Programmierstil!

Iteration

- *Unerreichbare Befehle* sind Anweisungen, die (u.a.)
  - hinter einer **`break`**- oder **`continue`**-Anweisung liegen, die ohne Bedingung angesprungen werden;
  - in Schleifen stehen, deren Testausdruck zur Compile-Zeit **`false`** ist.
- Solche unerreichbaren Anweisungen sind in Java nicht erlaubt, sie werden vom Compiler nicht akzeptiert.
- Einzige Ausnahme sind Anweisungen, die hinter der Klausel

  ```
  if (false)
  ```

  stehen.
- Diese Anweisungen werden von den meisten Compilern nicht in den Bytecode übernommen, sondern einfach entfernt.
- Man spricht dann von *bedingtem Kompilieren*.

Kapitel 9: Strukturierung von Programmen i

Module

- Klassen (so wie wir sie bisher kennen: mit statischen Methoden und Klassenvariablen) sind ausführbar, wenn sie eine `main`-Methode enthalten.
- Klassen können aber auch keine `main`-Methode enthalten, dann stellen sie verschiedene Algorithmen (in Form anderer statischer Methoden) und globale Größen wie Konstanten zur Verfügung.
- Letztere Variante hatten wir Modul genannt.
- Wir hatten ebenfalls diskutiert, dass Module ein wichtiges Strukturierungskonzept darstellen.
- Funktionalitäten (insbesondere Methoden), die semantisch zusammen gehören (also z.B. in einem engen Kontext der realen Welt stehen), kann man gut in einem Modul bündeln.

Pakete

- Bei großen Programmen entstehen allerdings viele Klassen/Module (und sog. Schnittstellen, die wir später kennenlernen).
- Um einen Überblick über diese Menge zu bewahren, wird ein zusätzliches Strukturierungskonzept benötigt, das von den Details abstrahiert und die übergeordnete Struktur der Module/Klassen verdeutlicht.
- Ein solches weiteres, übergeordnetes Strukturierungskonzept stellen in Java die *Pakete* (*packages*) dar.

Pakete

- Packages erlauben es, Komponenten zu größeren Einheiten zusammenzufassen.
- Die meisten Programmiersprachen bieten diese Strukturierungskonzepte (Module bzw. Packages, teilweise unter anderen Namen) an.
- Packages gruppieren Klassen, die einen gemeinsamen Aufgabenbereich. haben.

Packages in Java

- In Java können Klassen zu Packages zusammengefasst werden.
- Packages dienen in Java dazu,
  - große Gruppen von Klassen, die zu einem gemeinsamen Aufgabenbereich gehören, zu bündeln,
  - potentielle Namenskonflikte zu vermeiden,
  - Zugriffe und Sichtbarkeit zu definieren und kontrollieren,
  - eine Hierarchie von verfügbaren Komponenten aufzustellen.

Packages in Java

- Jede Klasse in Java ist Bestandteil von genau einem Package.
- Ist eine Klasse nicht explizit einem Package zugeordnet, dann gehört es implizit zu einem *Default*-Package.
- Packages sind hierarchisch gegliedert, können also Unterpackages enthalten, die selbst wieder Unterpackages enthalten, usw.
- Die Package-Hierarchie wird durch Punktnotation ausgedrückt:

  ```
  p1.p11.p111. ... .Klasse
  ```

- Der vollständige Package-Name beinhaltet alle Ober-Packages.

Import von Packages

- Der vollständige Name einer Klasse besteht wie bereits angedeutet aus dem Klassen-Namen *und* dem Package-Namen: `packagename.KlassenName`
- Package-Namen bestehen nach Konvention immer aus Kleinbuchstaben.
- Um eine Klasse verwenden zu können (z.B. die Methode `methodeX()` aus der Klasse `KlasseX` im Package `packageX`), muss angegeben werden, in welchem Package sie sich befindet.

Import von Packages

- Dies kann auf zwei Arten geschehen:
  1. Die Klasse wird an der entsprechenden Stelle im Programmtext über den vollen Namen angesprochen:
     ```
     packageX.KlasseX.methodeX();
     ```
  2. Am Anfang des Programms werden die gewünschten Klassen mit Hilfe einer **`import`**-Anweisung eingebunden (geladen):
     ```
     import packageX.KlasseX;
     ...
     methodeX();
     ```
- Achtung: werden zwei Klassen gleichen Namens aus unterschiedlichen Packages importiert, müssen die Klassen trotz **`import`**-Anweisung mit vollem Namen aufgerufen werden!

Import von Packages

- Klassen des Default-Packages können ohne explizite **`import`**-Anweisung bzw. ohne vollen Namen verwendet werden.
- Wird in der **`import`**-Anweisung eine Klasse angegeben, wird genau diese Klasse importiert.
- Will man alle Klassen eines Packages auf einmal importieren, kann man dies mit der folgenden **`import`**-Anweisung tun:

  `import` `packagename.*;`
- Achtung: es werden dabei wirklich nur die Klassen aus dem Package `packagename` eingebunden und nicht etwa auch die Klassen aus Unter-Packages von `packagename`

Packages aus der Java-Klassenbibliothek

- Die Java-Klassenbibliothek bietet bereits eine Vielzahl von Klassen an, die alle in Packages gegliedert sind.
- Beispiele für vordefinierte Packages:

`java.io`	Ein- und Ausgabe
`java.util`	nützliche Sprach-Werkzeuge
`java.awt`	Abstract Window Toolkit
`java.lang`	Elementare Sprachunterstützung

  usw.
- Die Klassen im Package `java.lang` sind so elementar (z.B. enthält `java.lang` die Klasse `System`), dass sie von jeder Klasse automatisch importiert werden. Ein expliziter Import mit **`import`** `java.lang.*;` ist also nicht erforderlich.

Deklaration eigener Packages

- Ein eigenes Package `mypackage` wird angelegt, indem man vor eine Klassendeklaration und vor den **`import`**-Anweisungen die Anweisung **`package`** `mypackage;` platziert.
- Es können beliebig viele Klassen (jeweils aber mit unterschiedlichen Namen) mit der Anweisung **`package`** `mypackage;` im selben Package gruppiert werden.

Deklaration eigener Packages

- Wie Sie wissen, muss die Deklaration einer Klasse `X` in eine Datei `X.java` geschrieben werden.
- Darüberhinaus müssen alle Klassendeklarationen (also die entsprechenden `.java`-Dateien) eines Packages `p` in einem Verzeichnis `p` liegen.
- Beispiel:
  - Die Datei `Klasse1.java` mit der Deklaration der Klasse `package1.Klasse1` liegt im Verzeichnis `package1`.
  - Die Datei `Klasse2.java` mit der Deklaration der Klasse `package1.unterpackage1.Klasse2` liegt im Verzeichnis `package1/unterpackage1`.

Zugriffsrechte und Sichtbarkeit

- Wir hatten bereits ein Schlüsselwort zur Spezifikation der Sichtbarkeit von Klassen und Klassen-Elementen (globale Größen/statische Methoden) kennengelernt: **`public`**.
- Klassen und Elemente mit der Sichtbarkeit **`public`** sind von allen anderen Klassen (insbesondere auch Klassen anderer Packages) sichtbar und zugreifbar.
- Darüberhinaus gibt es weitere Möglichkeiten, eine davon ist **`private`**.
- Klassen und Elemente mit der Sichtbarkeit **`private`** sind nur innerhalb der eigenen Klasse (also auch *nicht* innerhalb möglicher Unterklassen oder Klassen des selben Packages) sichtbar und zugreifbar.

Zugriffsrechte und Sichtbarkeit

- Klassen und Elemente, deren Sichtbarkeit *nicht* durch ein entsprechendes Schlüsselwort spezifiziert ist, erhalten per Default die sogenannte *package scoped* (*friendly*) Sichtbarkeit: diese Elemente sind nur für Klassen innerhalb des selben Packages sichtbar und zugreifbar.
- Es gibt dann noch eine vierte Möglichkeit (**`protected`**), die wir mal wieder erst später kennen lernen.