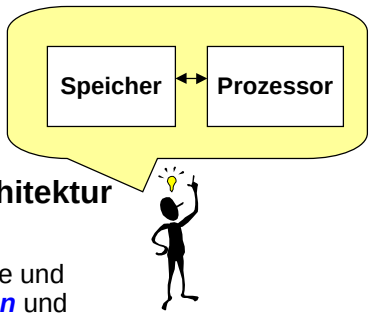
Imperative Programmierung

- **Konzepte der imperativen Programmierung**
- **Variable und Wertzuweisung**
- **Prozeduren**
- **rekursive Prozeduren**
- **Parameterübergabe**
- **Gültigkeitsbereich und Lebensdauer**

Modell der imperativen Programmierung

- **Synomym:**
  - Befehls-orientierte Programmierung

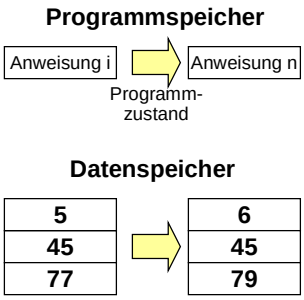


- **Geprägt durch die von-Neumann-Architektur**
  - Die CPU führt ***Maschinenbefehle*** aus
  - Deshalb müssen über den sog. Bus Befehle und Daten vom Speicher in die CPU ***übertragen*** und die Ergebnisse ***rückübertragen*** werden.
- **Mit imperativen Programmiersprachen setzen wir Entwürfe um:**
  - ***Aktionen*** fassen Folgen von Maschinenbefehlen zusammen,
  - ***Variable*** abstrahieren vom physischen Speicherplatz.

Konzepte der imperativen Programmierung

Semantik eines imperativen Progamms

- **Wesentliches Merkmal eines Programms**
  - ***Zustand*** der Daten im Speicher
  - Stand des Befehlszählers
- **Semantik eines Befehls**
  - Übergang von ZUSTAND1 -> ZUSTAND2
- **Programmierer beschreibt einen Prozeß von Zustandsübergängen**
  - durch elementare Anweisungen
  - durch den Kontrollfluß (Programmpfad)
- ***Variable*** **und** ***Wertzuweisung*** **modellieren Zustand und Zustands-übergang im Datenspeicher**



Konzepte der imperativen Programmierung

Imperatives Programmieren

- **Aufgaben des Programmierers**
  - Planung der ***Speicherbelegung***
    - Welche Daten braucht mein Programm?
  - Planung der ***Unterprogramme***
    - Aus welchen Funktionen und Prozeduren soll das Programm bestehen?
    - Wie sollen diese die Werte der Daten verändern?
  - Planung des ***Kontrollflusses***
    - In welcher Reihenfolge sollen die Operationen ausgeführt werden?
  - Planung des ***Datenflusses***
    - Welche Daten müssen von welchen Operationen an andere übergeben werden?

Deklaration von Daten

Deklaration von Unterprogrammen

Anweisungen

Reihenfolge in Programmpfad bzw. Parameterübergabe

Konzepte der imperativen Programmierung

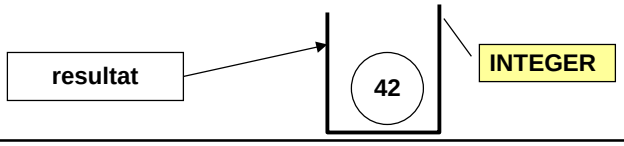
Objekte und Aktionen

- **Wir unterscheiden bei Datenobjekten:**
  - ***Konstante***: Objekte, deren Wert während der Ausführung des Algorithmus unverändert bleibt.
  - ***Variable***: Objekte, deren Wert sich während der Ausführung des Algorithmus verändern kann.
  - Für beide gilt, daß ihre Werte einen ***Typ*** haben. Diese sind zunächst die elementaren Datentypen.
- **Wir unterscheiden bei Aktionen:**
  - Veränderung des Werts einer Variablen durch ***Zuweisung***.
  - Festlegung der nächsten Aktion durch den sog. ***Kontrollfluß*** (Ablaufsteuerung).

Variable und Wertzuweisung

Charakterisierung

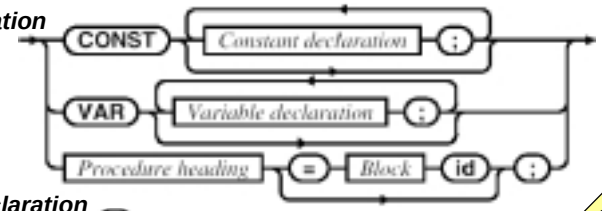
- **Variable**
  - Speicherplatz mit seinem Wert
  - besitzt einen **Namen**, unter dem man ihn ansprechen kann
  - bei Sprachen mit Typsystem muß jede Variable einem **Datentyp** zugeordnet sein
  - Datentyp legt fest, welche **Struktur** und **Werte** eine Variable besitzt/ annehmen kann und wie Werte im Programm als Literale/Aggregate hingeschrieben werden
  - Ferner legt ein Datentyp fest, welche **Operationen** ausgeführt werden dürfen
  - Variablen werden im Deklarationsteil von Programmeinheiten (Blöcke, Module) vereinbart (deklariert)
  - Variable kann als Behälter betrachtet werden: hat Wert und Typ



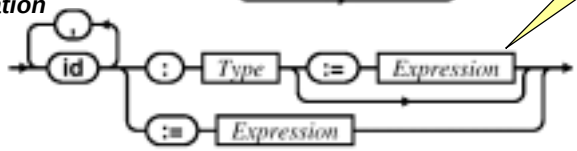
Variable und Wertzuweisung

Deklarationen (erweitert)

Declaration



Variable declaration



Initialisierung der Variablen

Syntaktisch korrekt

Laufzeitfehler?

```
VAR r : INTEGER := 4;
    x : INTEGER := (4 * 8 -2);

    b := FALSE;
    c := TRUE;
    d := (b = c);

    y : REAL;
```

```
VAR x  : INTEGER := 4;
    xx : INTEGER := (2 * x);
    y  : INTEGER := (4 DIV y);
```


Variable und Wertzuweisung

Wertzuweisung

- **Wertzuweisung**
  - dient dazu, den Wert einer Variablen zu verändern
  - Syntax (vereinfacht): **Variable := Ausdruck**
  - Der Ausdruck wird zuerst ausgewertet, das Ergebnis wird anschließend der Variablen zugewiesen
    - In diesem Zusammenhang sprechen wir oft von der rechten und der linken Seite einer Zuweisung:
    - LH value, RH value
  - Als Ausdruck auf der rechten Seite verwenden wir meist arithmetische und Boolesche Ausdrücke, Vergleiche und Zeichen bzw. Zeichenketten.
  - ***Typkompatibilität***:
    Der Typ des Bezeichners muß zum Typ des Ausdrucks passen, d.h. zunächst, die Typen müssen gleich sein.

Variable und Wertzuweisung

Beispiele: Wertzuweisung

- **Deklaration**
  - `VAR x, y, z: CARDINAL;`
- **einige (korrekte und inkorrekte) Wertzuweisungen für x:**
  -
```
x :=  0;
x := MAX (CARDINAL);
x :=  y;   (* sicher richtig *)
```
  -
```
x := -1;
x := MAX (CARDINAL)+1;
x := -y-1; (* sicher falsch  *)
```
  -
```
x := y+z; x := z-y;(* richtig oder falsch,  *)
                        (* je nach Wert von y, z *)
```


Variable und Wertzuweisung

Beispiel: Wertzuweisung

```
MODULE Vertausche1 EXPORTS Main;
(* Vertauscht zwei eingebenene Werte *)

IMPORT SIO;

VAR x, y, h : INTEGER;

BEGIN
  x := SIO.GetInt();
  y := SIO.GetInt();

  h     := x;
  x     := y;
  y     := h;

  SIO.PutText("x = "); SIO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); SIO.PutInt(y);
END Vertausche1.
```

Deklaration der Variablen

Initialisierung der Variablen

Variable und Wertzuweisung

Diskussion der RETURN-Anweisung

```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
  VAR min: INTEGER;
BEGIN
  IF m <= n THEN
    min := m;
  ELSE
    min := n
  END;
  RETURN min;
END Minimum ;
```

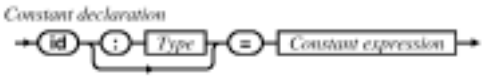
```
PROCEDURE Minimum (m,n: INTEGER) : INTEGER =
BEGIN
  IF m <= n THEN
    RETURN m
  ELSE
    RETURN n
  END;
END Minimum ;
```

- **Anmerkung:**
  - mehrere RETURN-Anweisungen machen eine Funktion unübersichtlich
- **Empfehlung**
  - Code-Effizienz gegen Lesbarkeit abwägen!

Variable und Wertzuweisung

Symbolische Konstanten

- **Verwendung von Zahlen-Konstanten ("Literalen") in Ausdrücken führt zu Problemen bei der Wartung!**
- **Konstante**
  - Bezeichner mit einem ***festen*** Wert
  - hat einen Datentyp
  - muß deklariert werden
  - überall, wo der Konstantenbezeichner auftritt, wird der Konstantenwert eingesetzt
  - Nach der Deklaration kann ihr ***kein Wert zugewiesen*** werden



- **Beispiel:**

```
CONST PI = 3.141;
VAR umfang, radius : REAL;

…

umfang := 2 * PI * radius
```

```
CONST
  Arbeitstage   = 5;
  Arbeitszeit   = 8;
  Wochenstunden = Arbeitstage *
                  Arbeitszeit;
```

Prozeduren

Der Prozedurbegriff

- **Prozedur ist ein zentraler Begriff der prozeduralen Programmierung.**
- **Fachlich ist eine Prozedur**
  - die programmiersprachliche ***Realisierung*** eines Algorithmus.
- **Softwaretechnisch**
  - kann eine Prozedur zunächst als ***benannte Anweisungsfolge*** verstanden werden.
- **Die Grundidee ist,**
  - den Namen der Prozedur "***stellvertretend***" für diese Anweisungsfolge zu verwenden.
  - Die Parameter erlauben die Prozedur mehrfach zu nutzen

Prozeduren

Algorithmische Abstraktion

- **Die Prozedur ist eine wesentliche Umsetzung des Konzepts der *algorithmische Abstraktion* (auch *Prozeßabstraktion* genannt):**
  - Statt einer expliziten Anweisungsfolge (der genauen Verarbeitungsvorschrift) wird ein davon ***abstrahierender*** Name verwendet.
- **Abstraktion wird hier sowohl als *Vorgang* als auch als *Ergebnis des Vorgangs* verstanden:**
  - Im Vorgang der algorithmischen Abstraktion sehen wir von der konkreten Anweisungsfolge ab und bringen diese auf *"**einen Begriff**"*.
  - Das Ergebnis ist eine Entwurfs- und ***Programmeinheit*** – die Prozedur.

Prozeduren

Beispiel: Abstraktionsprozeß

```
Programm Telefonieren
  Hörer_abheben;
  Telefonnummer_wählen;
  Gespräch_führen;
  Hörer_auflegen;
ENDE Telefonieren.
```

Algorithmische Abstraktion durch **Prozeduren**: Statt einer Anweisungsfolge wird ein **Name** verwendet.

Ergebnis
Programmeinheit

```
Prozedur Telefonnummer_wählen
 IF Telefonnummer_gespeichert
THEN
   Kurzwahltaste_drücken
 ELSE
   Telefonnummer_eintippen
 END
ENDE Telefonnummer_wählen.
```


Prozeduren

Kennzeichen von Prozeduren

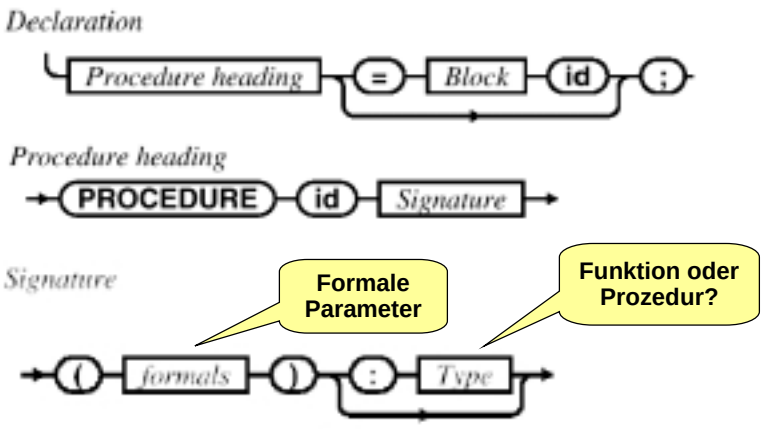
- **Um den Prozedurbegriff verstehen zu können, benötigen wir drei Begriffe:**
- **Parametrisierung:**
  - Der Mechanismus zum ***Datenaustausch*** zwischen Prozedur und Umgebung.
- **Sichtbarkeit:**
  - Der Programmbereich, in dem ***Namen bekannt*** sind.
- **Lebensdauer:**
  - Der ***Zeitraum***, in dem die Werte von Programmobjekten zugegriffen werden können.

Prozeduren vs. Funktionen

- **Wurden bisher zur Ausgabe verwendet**
  - SIO.PutText ("Hallo")
- **Sind Funktionen "ähnlich"**
  - werden mit PROCEDURE eingeleitet, können auch rekursiv sein
- **Unterschied zu Funktionen**
  - Prozeduren liefern ***kein*** Ergebnis im Sinne eines Funktionsergebnisses!
  - Konsequenz:
    - Prozeduren besitzen keinen ***Ergebnistyp***
    - Aufruf einer Prozedur ist kein Ausdruck, sondern eine Anweisung
- **Zweck einer Prozedur**
  - ***Zusammenfassen*** einer "Funktionalität" (im Sinne der Lokalität)
  - ***Verändern*** der ihr übergebenen Parameter
  - Durchführung einer ***Nebenwirkung*** (z.B. Ausgabe einer Meldung)

Prozedurdeklaration

- **Syntax:**



- **Beispiele:**

```
PROCEDURE PutChar(ch: CHAR; wr: Writer := NIL) = ...

PROCEDURE Minimum(n,m : INTEGER): INTEGER = ...
```

Prozeduren

Prozeduraufruf: Syntax

- **Beim Aufruf einer Prozedur werden die aktuellen Parameter an die formalen übergeben.**
- **Zur Übersetzungszeit wird überprüft:**
  - Der Name im Aufruf muß ***gleich*** dem Prozedurnamen sein.
  - Die ***Anzahl*** der aktuellen Parameter muß gleich der Anzahl der formalen sein.
  - Die Bindung der jeweiligen Parameter wird entsprechend ihrer ***Position*** im Aufruf und in der Prozedurdeklaration vorgenommen.
  - Die aktuellen Parameter müssen ***typkompatibel*** zu den formalen Parametern sein (d.h. meist typgleich).

```
PROCEDURE Minimum ( m, n : INTEGER) : INTEGER = …


res := Minimum (x, y); (* korrekter Aufruf)
res := Mini (x, y);
res := Minimum (x, y, z);
```

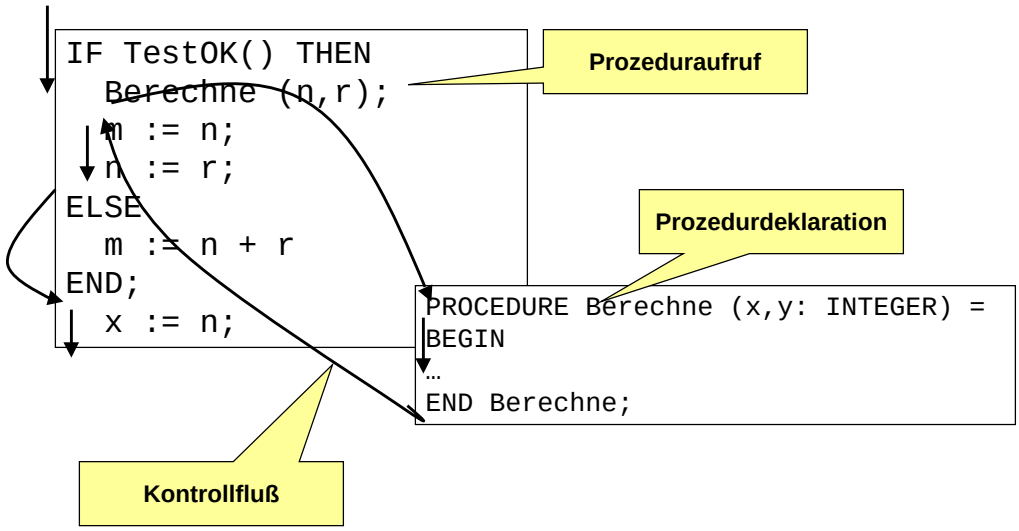

Prozeduren

Prozeduraufruf: Semantik

- **Der Prozeduraufruf ist die explizite Anweisung,**
  - daß die Prozedur ***ausgeführt*** werden soll.
- **Eine Prozedur ist *aktiv*,**
  - nachdem sie gerufen wurde und in der Ausführung ihrer Anweisungen noch kein vordefiniertes Ende erreicht hat.
- **Für den Prozeduraufruf in imperativen Sprachen ist charakteristisch:**
  - Beim Aufruf wechselt die ***Kontrolle*** (d.h. die Abarbeitung von Anweisungen) vom Rufer zur Prozedur.
  - Dabei werden die aktuellen Parameter an die formalen ***gebunden***.
  - Prozeduren können wieder Prozeduren aufrufen. Dabei wird der Aufrufer unterbrochen (suspended), so daß die Kontrolle stets bei einer Prozedur ist.
  - Nach der Ausführung der Prozedur kehrt die Kontrolle zum Rufer zurück; die Ausführung wird mit der ***Anweisung nach dem Aufruf*** fortgesetzt.

Prozeduren

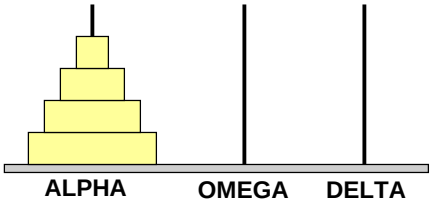
Beispiel: Prozeduraufruf



Rekursive Prozeduren

Rekursion: Aufgabe

- **Aufgabe: Türme von Hanoi**
  - bewege die Scheiben des Turms von ALPHA nach OMEGA
  - es darf immer ***nur eine Scheibe*** bewegt werden
  - niemals darf eine Scheibe auf eine kleinere bewegt werden



- **Lösungsstrategie (Divide & Conquer)**
  - allgemeine Lösung für einen Turm der Höhe h von ALPHA nach OMEGA
    - h = 0 gar nichts machen
    - h > 0 1. Turm der Höhe h-1 von ALPHA nach DELTA über OMEGA
      2. Scheibe von ALPHA nach OMEGA legen
      3. Turm der Höhe h-1 von DELTA nach OMEGA über ALPHA

Rekursive Prozeduren

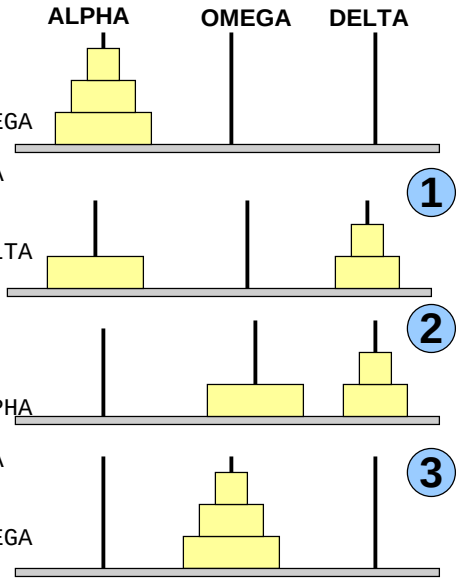
Rekursion: Programmtext

```
MODULE Hanoi EXPORTS Main;
(* Ausgabe der Zugfolge fuer Tuerme von Hanoi *)

IMPORT SIO;

PROCEDURE DruckeZug (hoehe: CARDINAL; von, nach : TEXT)  =
BEGIN
  SIO.PutText ("Scheibe "); SIO.PutInt (hoehe);
  SIO.PutText (" von " & von & " nach " & nach); SIO.Nl();
END DruckeZug ;

PROCEDURE BewegeTurm ( hoehe : CARDINAL; von, nach, ueber: TEXT)  =
BEGIN
  IF hoehe > 0 THEN
    BewegeTurm (hoehe-1, von, ueber, nach);
    DruckeZug  (hoehe, von, nach);
    BewegeTurm (hoehe-1, ueber, nach, von);
  END;
END BewegeTurm;

BEGIN
  BewegeTurm(SIO.GetInt(), "ALPHA", "OMEGA", "DELTA" );
END Hanoi.
```


Rekursive Prozeduren

Rekursion: Laufzeitgeschehen

```
3 ALPHA OMEGA DELTA
  ❶ 2 ALPHA DELTA OMEGA
    ❶ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
    ❷ Scheibe 2 von ALPHA nach DELTA
    ❸ 1 OMEGA DELTA ALPHA
      ❶ 0 OMEGA ALPHA DELTA
      ❷ Scheibe 1 von OMEGA nach DELTA
      ❸ 0 ALPHA DELTA OMEGA
  ❷ Scheibe 3 von ALPHA nach OMEGA
  ❸ 2 DELTA OMEGA ALPHA
    ❶ 1 DELTA ALPHA OMEGA
      ❶ 0 DELTA OMEGA ALPHA
      ❷ Scheibe 1 von DELTA nach ALPHA
      ❸ 0 OMEGA ALPHA DELTA
    ❷ Scheibe 2 von DELTA nach OMEGA
    ❸ 1 ALPHA OMEGA DELTA
      ❶ 0 ALPHA DELTA OMEGA
      ❷ Scheibe 1 von ALPHA nach OMEGA
      ❸ 0 DELTA OMEGA ALPHA
```



Parameter-übergabe

Parameter

- **Bisher**
  - Funktionen besitzen ausnahmslos ***Eingabeparameter***
  - Wert dieser Parameter kann nicht ***geändert*** werden
- **Allgemein gibt es folgende Parameterarten für Prozeduren**
  - ***Eingabeparameter***
    - vor dem Aufruf wird der aktuelle Parameter ausgewertet und dem formalen Parameter zugewiesen (***call-by-value***)
  - ***Ausgabeparameter***
    - dienen dazu, Ergebnisse einer Prozedur an den Aufrufer zurückzugeben
    - Wert ist zum Zeitpunkt des Aufrufs undefiniert (***call-by-reference***)
  - ***Ein- / Ausgabeparameter (Transienten)***
    - vereinen Eigenschaften beider Arten
- **Modula-3 nutzt dazu zwei Parameterübergabemechanismen**

Parameter-übergabe

Übergabemechanismen: Call by Value

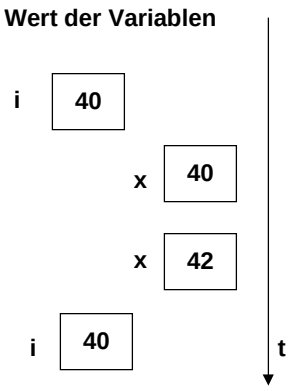
- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Wertparameter***
    - realisieren Eingangsparameter
    - Der aktuelle Parameter muß ein ***Ausdruck*** sein (Spezialfall: Variable, d.h. Bezeichner für ein Objekt).
    - Beim Aufruf der Prozedur wird ein dem Typ des formalen Parameters entsprechendes ***lokales Objekt*** angelegt. Ist der aktuelle Parameter eine Variable, so entsteht dabei eine ***Kopie*** des Parameter-Objekts.
    - In jedem Falle wird der Wert des aktuellen Parameters ***berechnet*** und dem formalen Parameter (-Objekt) zugewiesen.
    - Veränderungen des formalen Parameters in der Prozedur haben nur ***lokale Auswirkung***. Der aktuelle Parameter bleibt unverändert.
    - Schlüsselwort ***VALUE*** zeigt einen Wertparameter an
    - steht kein Schlüsselwort, ist es ***per default*** ein Wertparameter

Parameter-übergabe

Beispiel: Call by Value

```
PROCEDURE Proc1 (VALUE x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Nichts passiert")
ELSE
  SIO.PutLine ("kein Call by Value")
END;
```



Parameter-übergabe

Übergabemechanismen: Call by Reference

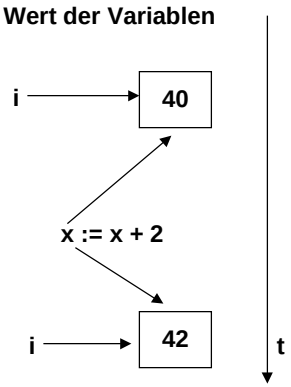
- **Der formale Parameter beim *Call by Value* ist ein**
  - ***Variablenparameter*** (oder Referenzparamter)
    - realisieren Ausgangs und Ein- / Ausgangsparameter
    - Der aktuelle Parameter muß ein ***Bezeichner für ein Objekt*** sein.
    - Beim Aufruf wird der formale Parameter durch einen ***Verweis*** auf den aktuellen Parameter ersetzt.
      D.h. der formale Parameter wird als lokaler Bezeichner für das aktuelle Parameterobjekt ***substituiert***.
    - Jede Änderung des formalen Parameters ist ***direkt*** im aktuellen Parameter wirksam.
    - Veränderungen des formalen Parameters in der Prozedur haben auf den aktuellen Parameter Auswirkung, d.h. Objekte im Namensraum des Rufers können ***verändert*** werden. Auf diese Weise können von einer Prozedur ***Ergebnisse*** zurückgegeben werden.
    - Schlüsselwort ***VAR*** zeigt einen Variablenparameter an

Parameter-übergabe

Beispiel: Call by Reference

```
PROCEDURE Proc1 (VAR x: INTEGER);
…
BEGIN
  x := x + 2;
  SIO.PutInt (x);
END Proc1;
```

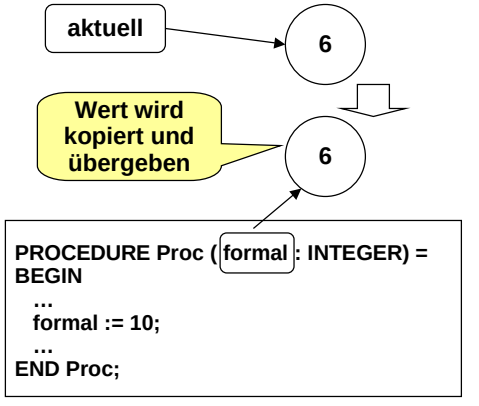
```
VAR i: INTEGER
…
i := 40;

Proc1 (i);

IF i = 40 THEN
  SIO.PutLine ("Call by Value")
ELSE
  SIO.PutLine ("Call by Reference")
END;
```

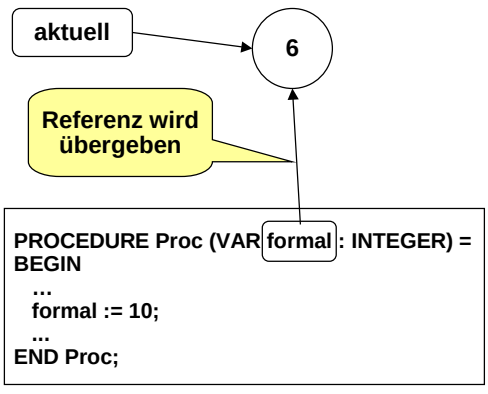


Parameter-übergabe

Call-by-value <-> Call-by-reference

```
VAR aktuell : INTEGER
…
aktuell := 6;
Proc (aktuell);
```





Parameter-übergabe

Beispiel: Wert- Variablenparameter

```
MODULE Parameter EXPORTS Main;
IMPORT IO, SIO;

VAR aktuell: INTEGER;

PROCEDURE Proc1 (VALUE formal : INTEGER) =
BEGIN
  formal := 10;
END Proc1;

PROCEDURE Proc2 (VAR formal : INTEGER) =
BEGIN
  formal := 10;
END Proc2;

BEGIN
  aktuell := 6;        Proc1 (aktuell);
  SIO.PutText("aktuell (Proc1) = "); IO.PutInt(aktuell); SIO.Nl();
  aktuell := 6;        Proc2 (aktuell);
  SIO.PutText("aktuell (Proc2) = "); IO.PutInt(aktuell);
END Parameter.
```

Wertparameter

Variablen-parameter

aktuell (Proc1) = 6
aktuell (Proc2) = 10

Ausgabe des Programms

Parameter-übergabe

Beispiel: Variablenparameter

```
MODULE Vertausche2 EXPORTS Main;
IMPORT IO, SIO;

PROCEDURE Vertausche (VAR w1, w2 : INTEGER) =
VAR hilfe : INTEGER := w1;
BEGIN
  w1    := w2;
  w2    := hilfe;
END Vertausche;

VAR x, y : INTEGER;

BEGIN
  x := IO.GetInt();
  y := IO.GetInt();
  Vertausche (x, y);
  SIO.PutText("x = "); IO.PutInt(x);  SIO.Nl();
  SIO.PutText("y = "); IO.PutInt(y);
END Vertausche2.
```

Vertauscht die Werte der beiden Parameter

Parameter-übergabe

Beispiel: Ausgabeparameter

- **Ausgabeparameter**
  - dient dazu, einen Wert an den Aufrufer zurückzugeben

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR  ergebnis : REAL;

PROCEDURE Quadrat ( VALUE x : REAL; VAR wert : REAL )=
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Ausgabeparameter oder Ein- Ausgabeparameter ?

Wert ist beim Aufruf undefiniert

Parameter-übergabe

Datenaustausch: Beispiel - 1

In einer Reihe von Meßwerten soll der **laufende Mittelwert** berechnet werden. Benötigte Objekte und Aktionen:

```
Wert, Mittelwert, Summe : REAL;
Anzahl : INTEGER;
(* BerechneMWert
      Addiere neuen Wert und Summe,
      Erhöhe Anzahl um 1,
      Mittelwert := Summe / Anzahl *)
```

Austausch über Parameter:

```
PROCEDURE BerechneMWert (     wert     : REAL;
                          VAR anzahl    : INTEGER;
                          VAR summe, mw : REAL) =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  mw := summe / anzahl;
END BerechneMWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;
BerechneMWert (wert,anzahl,summe,mw);
BerechneMWert (2*wert,anzahl,summe,mw);
```

Parameter-übergabe

Datenaustausch: Beispiel - 2

Datenaustausch über Funktionsergebnis:

```
PROCEDURE MWert (    wert  : REAL;
                 VAR anzahl: INTEGER;
                 VAR summe : REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl:= anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0; wert := 4.0;

SIO.PutReal (MWert(wert,anzahl,summe));
...
SIO.PutReal (MWert(2*wert,anzahl,summe));
```


Parameter-übergabe

Datenaustausch: Beispiel - 3

Datenaustausch über Funktionsergebnis:

```
VAR summe: REAL; anzahl: INTEGER;

PROCEDURE MWert (wert:REAL): REAL =
BEGIN
  summe := summe + wert;
  anzahl := anzahl + 1;
  RETURN summe / anzahl;
END MWert ;
```

Verwendung:

```
summe := 0.0; anzahl := 0;

SIO.PutReal (MWert(4.0);
...
SIO.PutReal (MWert(10.0));
```

Parameter-übergabe

Diskussion der Beispiele - 1

- **Beispiel 1:**
  - Die Verwendung von VAR-Parametern in einer Prozedur zur Rückgabe von Ergebnissen ist in der imperativen Programmierung üblich.
  - Sie führt oft zu ***Verständnisproblemen***, da sowohl in der Deklaration als auch in der Verwendung klar sein muß, was das eigentliche Ergebnisobjekt (hier: der Mittelwert) ist.
- **Beispiel 2:**
  - Die Verwendung einer Funktion zur Berechnung genau eines Wertes ist dann sauber, wenn alle anderen Parameter als ***Wertparameter*** verwendet werden.
  - Die Modellierung einer Zustandsveränderung (hier: Summe und Anzahl) außerhalb der Funktion mit Hilfe von VAR-Parametern ist ein ***Seiteneffekt***, der die ***Lokalität*** der Funktion zerstört.

Parameter-übergabe

Diskussion der Beispiele - 2

- **Beispiel 3:**
  - Die Verwendung von ***globalen*** Variablen, die in einer Prozedur oder Funktion als Seiteneffekt verändert werden, ist die ***schlechteste*** Lösung.
  - Zustandsveränderungen über globale Variablen, die in der Signatur der Prozedur nicht aufgeführt sind, sind unverständlich und extrem ***fehleranfällig***.
- **Funktionsprozeduren**
  - Eine Funktion kann in imperativen Sprachen nur dann sauber modelliert werden, wenn
    - alle Parameter als ***Wertparameter*** übergeben werden,
    - in der Funktion ***keine globalen Variablen*** verwendet werden.
  - In Funktionen sollte auch keine globalen Variablen ***lesend verwendet*** werden, da dies Funktionen an ihren Verwendungskontext ankoppelt.
  - Merke: Funktionen sollten "***in sich***" verständlich sein.

Parameter-übergabe

Regeln für die Parameterverwendung

- **Wertparameter sind Variablenparameter vorzuziehen**
  - Änderung an Parametern in Prozeduren ist häufig eine ***Fehlerquelle***, die schwer zu finden ist.
- **Variablenparameter sollten nur verwendet werden, wenn**
  - Prozedur-Ergebnisse ***übergeben*** werden sollen (Ausgabeparameter)
  - Kopieren des Parameters ***nicht möglich*** ist (bei gewissen Datenstrukturen)
  - Kopieren zu ***ineffizient*** ist (bei sehr großen Datenstrukturen)
- **Funktionen haben nur Wertparameter**
  - liefern ihr Ergebnis durch ihren ***Namen*** zurück
  - geht in Modula-3 nur, wenn
    - genau ein Wert zurückgegeben werden soll
  - Rückgabe durch Namen und Variablenparameter muß ***vermieden*** werden

Parameter-übergabe

Zusammenfassung: Prozeduren und Parameter

- **In imperativen Sprachen wird oft die Prozedur in den Vordergrund gestellt.**
- **Die Funktion ist gelegentlich (z.B. Modula-3) nur eingeschränkt verwendbar, kann dafür aber *Seiteneffekte* erzeugen (nicht wünschenswert).**
- **Nur durch eine saubere Definition der *Signaturen* von Routinen kann ein verständlicher und weiterverwendbarer Entwurf erreicht werden, d.h. vor allem, jeder Datenaustausch mit der Umgebung sollte *explizit* sein.**
- **Die Modellierung von Zuständen und ihrer Veränderung ist nur in Verbindung mit einem entsprechenden *Modulkonzept* softwaretechnisch sauber zu lösen.**

  Das lernen wir später!

Gültigkeitsbereich und Lebensdauer

Def. Gültigkeitsbereich und Lebensdauer

- **Gültigkeitsbereich (scope) eines Bezeichners**
  - der ***statische Teil*** des Programms, in dem der Bezeichner mit exakt ***gleicher Bedeutung*** verwendet werden darf
  - ***Sichtbarkeitsbereich*** ist Teil des Gültigkeitsbereichs
  - Gültigkeitsbereich/Sichtbarkeitsbereich wird durch den Compiler überwacht
- **Lebensdauer eines Objekts (Variable, Prozedur)**
  - bezieht sich auf den zur ***Programmlaufzeit*** belegten Speicherplatz
  - macht nur Sinn für Objekte, die Speicher belegen
    - Typen belegen keinen Speicher
- **Es ist wichtig, beide Begriffe klar zu unterscheiden!**

Gültigkeitsbereich und Lebensdauer

Gültigkeitsbereich: lokale Deklarationen

- **Regeln für die Gültigkeit von Bezeichnern;**
  - Alle in einer Prozedur oder einem Modul deklarierten Bezeichner sind in der gesamten Prozedur / im gesamten Modul gültig.
  - Das gilt auch für den textuell vor der Deklaration eines Bezeichers liegenden Bereich
  - Davon ausgenommen sind Prozedur- und Modulkopf

```
PROCEDURE Proc ( in : REAL )=

  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Hier können die Bezeichner X und C verwendet werden.

```
PROCEDURE Proc ( in : REAL )=

  CONST CC : C * C
  VAR X : INTEGER;
  CONST C : 100;

BEGIN
  ...
END Proc;
```

Korrekte Vorwärts-referenz

Gültigkeitsbereich und Lebensdauer

Gültigkeitsbereich: Erweiterung

- **Durch IMPORT kann der Gültigkeitsbereich von Bezeichnern auf das *importierende Modul* erweitert werden.**

```
INTERFACE SIO;
...
PROCEDURE GetReal …;

PROCEDURE PutReal ...;

PROCEDURE GetText ...;

PROCEDURE PutText ...;

…
END SIO;
```

```
MODULE QuadratM1 EXPORTS Main;

IMPORT SIO;
CONST eingabe = 2.5;
VAR   ergebnis : REAL;

PROCEDURE Quadrat … =
BEGIN
  wert := ( x * x );
END Quadrat;

BEGIN
  Quadrat (eingabe , ergebnis);
  SIO.PutReal (ergebnis);
END QuadratM1.
```

Qualifizierter Bezeichner muß verwendet werden.

Gültigkeitsbereich und Lebensdauer

Prozeduren als Namensraum - 1

- **Namensraum**
  - Eine Prozedur bildet gegenüber der (textuellen) Umgebung ihrer Deklaration einen eigenen ***Namensraum***, d.h. sie kann lokale Objekte benennen und verwalten.
  - Die Namen der formalen Parameter sowie die im Deklarationsteil des Prozedurrumpfs deklarierten Bezeichner sind nur ***im Prozedurrumpf gültig***, d.h. bekannt.
  - In einer Prozedur können Bezeichner der Umgebung ***neu lokal*** deklariert werden; diese Bezeichner ***verdecken*** die global deklarierten Bezeichner, die zugehörigen Objekte sind in der Prozedur ***nicht sichtbar***.
- **Lebensdauer**
  - Die Lebensdauer von prozedurlokalen Objekten entspricht dem Zeitraum, in dem der Aufruf der Prozedur abgearbeitet wird. Sie werden zu Beginn der Prozedurausführung angelegt.
  - Werte gehen ***verloren***, wenn die Ausführung der Prozedur beendet ist.

Gültigkeitsbereich und Lebensdauer

Prozeduren als Namensraum - 2

```
PROCEDURE Proc ()=

  VAR x : INTEGER;

  PROCEDURE Proc1 (x :Integer) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Proc;
```

G2

G3

G1

- **Module und Prozeduren definieren eigene Namensräume**
  - Prozeduren können verschachtelt werden
  - Hierarchie von Gültigkeits-bereichen (Namensräumen)
- **Überlappung von Gültigkeitsbereichen**
  - Liegt innerhalb des Gültigkeitsbereiches G1 von X mit der Bedeutung B1 ein weiterer Gültigkeitsbereich G2 von X mit der Bedeutung B2, so handelt es sich um zwei ***verschiedene*** Objekte
  - innerhalb von G2 gilt nur B2, alle anderen Bedeutungen sind ***unsichtbar***.

Gültigkeitsbereich und Lebensdauer

Lebensdauer - 1

- **Durch Ausführung einer Prozedur P entsteht eine *Inkarnation*.**
- **Zur Inkarnation gehören *zur Laufzeit*:**
  - ein ***Ausführungspunkt*** (also ein Zeiger auf den gerade auszuführenden oder ausgeführten Befehl)
  - ***Speicherplätze*** für alle Bezeichner von Variablen und Wertparameter
  - ***Bezüge*** auf die konkreten Variablenparameter.
- **Informationen existieren bis zum Ende der Ausführung von P.**
- **Beispiel**
  - `PROCEDURE Test (ch: CHAR; VAR x:INTEGER)=`
    `VAR y: REAL`
- **`Test ('a', z)` führt zu einer Inkarnation mit**
  - Speicherplatz für ch und y
  - unter dem lokalen Bezeichner `x` einen Bezug (einer Referenz) auf z
  - nach Abschluß werden diese Speicherplätze wieder freigegeben

Gültigkeitsbereich und Lebensdauer

Lebensdauer - 2

- **Bemerkungen:**
  - **Sei M ein Modul,**
    - dann wird ***Speicherplatz*** für die Variablen permanent für die gesamte Laufzeit des Programms reserviert.
    - Eine eigentliche Inkarnation wird nur von dem Rumpf des Moduls gebildet; dieser hat weder Variablen noch Parameter.
  - Zu irgendeinem Zeitpunkt existieren i.a. neben der Inkarnation des ablaufenden Moduls ***Inkarnationen verschiedener*** Prozeduren, bei **rekursiven** Aufrufen auch mehrere der gleichen Prozedur.
  - Nur in der ***jüngsten*** aller existierenden Inkarnationen wandert der Ausführungspunkt weiter; diese wird als erste beendet.
  - Die Lebensdauer einer Variablen ist ***identisch*** mit der Existenz der zugehörigen Prozedur-Inkarnation.
  - Zu Beginn der Lebensdauer ist der Wert einer Variablen ***undefiniert***, darf also nicht verwendet werden.

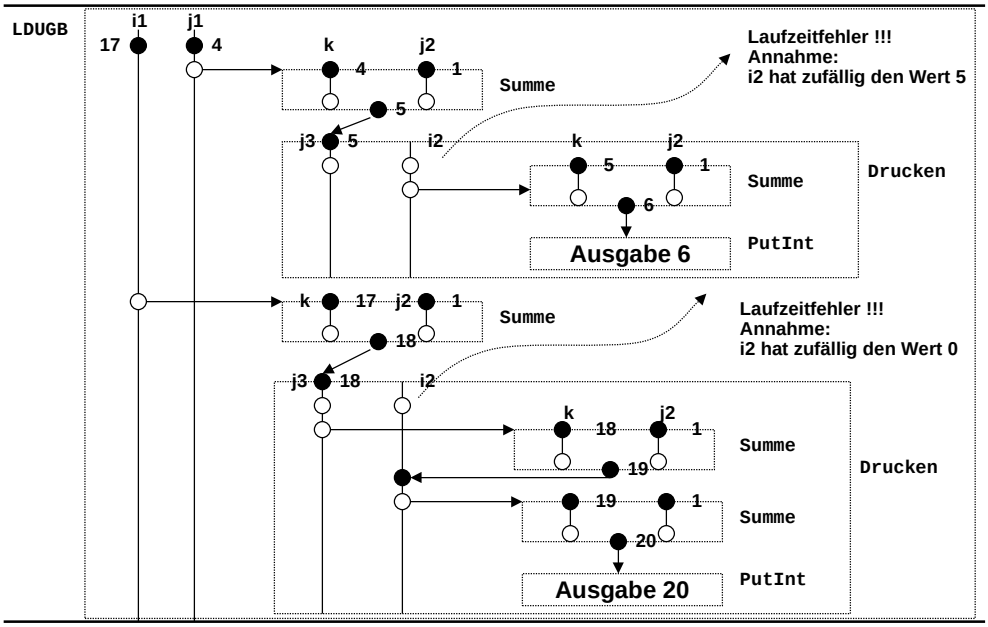
Gültigkeitsbereich und Lebensdauer

Beispiel: Lebensdauer - 1

```
MODULE LDUGB EXPORTS Main;
IMPORT SIO;
VAR i  , j  : INTEGER;

PROCEDURE Summe (k  : INTEGER) : INTEGER =
VAR j  : INTEGER;
BEGIN
  IF i  <10 THEN j  := 10 ELSE j  :=1 END;
  RETURN j  + k  ;
END Summe;

PROCEDURE Drucken (j  : INTEGER) =
VAR i  : INTEGER;
BEGIN
  IF i   #   j   THEN i  := Summe(j  ) END;
  SIO.PutInt (Summe(i  )); SIO.Nl();
END Drucken;

BEGIN
  i  := 17; j  := 4;
  Drucken (Summe(j  ));
  Drucken (Summe(i  );
END LDUGB.
```

Gültigkeitsbereich und Lebensdauer

Beispiel: Lebensdauer - 2



Gültigkeitsbereich und Lebensdauer

Terminologie

- **"Objekt" bezeichnet alles,**
  - was durch einen Bezeichner eingeführt wird (Modul, Prozedur, Konstante, Typ, Variable, Parameter),
  - keine Objekte sind demnach Operatoren oder Wortsymbole (z.B. VAR, END)
- **Ein Objekt heißt lokal in Block B,**
  - wenn es im Block B deklariert ist.
- **Ein Objekt heißt global,**
  - wenn es auf Modulebene deklariert ist.
- **Ein Objekt heißt global relativ zu B,**
  - wenn es in B gültig ist, aber nicht lokal in B ist

Gültigkeitsbereich und Lebensdauer

Beispiel: lokal, global, global relativ

```
MODULE Gueltigkeit EXPORTS Main;
  IMPORT SIO;
  VAR x : INTEGER;

  PROCEDURE Proc1 (x: INTEGER) =
  BEGIN
    x := x - 1;
    SIO.PutInt(x * x);
  END Proc1;

  PROCEDURE Proc2 (y: INTEGER) =
  BEGIN
    x := x + 1;
    SIO.PutInt(x * y);
  END Proc2;

BEGIN
  x := 5;
  Proc1 (x); SIO.Nl();
  Proc2 (x); SIO.Nl();
  SIO.PutInt(x);
END Gueltigkeit.
```

x ist global sichtbar im Modul

x ist lokal in der Prozedur Proc1

x ist global relativ in der Prozedur Proc2

y ist lokal in der Prozedur Proc2

Gültigkeitsbereich und Lebensdauer

Lokalität

- **Ziel**
  - Programme sollten mit ***möglichst wenig*** Aufwand korrigier- und modifizierbar sein.
- **Strategie**
  - ***hohe Lokalität*** durch enge Gültigkeitsbereiche.
  - Größtmögliche Lokalität ist daher ***vorrangiges Ziel*** einer guten Programmierung!
- **Ein Programm sollte dafür folgende Merkmale aufweisen:**
  - Die auftretenden Programmeinheiten (Prozeduren, Funktionen, Hauptprogramm) sind ***überschaubar***.
  - Die Objekte sind so ***lokal*** wie möglich definiert, jeder Bezeichner hat nur eine ***einzige***, bestimmte Bedeutung.
  - Die ***Kommunikation*** zwischen Programmeinheiten erfolgt vorzugsweise über eine möglichst kleine Anzahl von Parametern, nicht über globale Variable.

Gültigkeitsbereich und Lebensdauer

Vorteile lokaler Variabler

- **Lokale Variablen haben softwaretechnisch einige Vorteile.**
  - Deklaration und Verwendung stehen in einem ***textlichen*** Zusammenhang. Das erhöht die Lesbarkeit.
  - Die ***unfreiwillige*** Verwendung von globalen Variablen wird vermieden, d.h. globale Variablen müssen nicht vollständig bekannt sein.
  - Der Speicherverbrauch durch Prozeduren wird ***minimiert***, da Speicher für lokale Variablen nur während ihrer Aktivierung vorgehalten werden muß.
  - Lokalität:
    - Variablen sollen ***nur in dem Kontext*** deklariert werden, wo sie bekannt sein müssen. Eine Verteilung erschwert die Änderbarkeit.
  - Kapselung:
    - Außerhalb eines Kontextes (Modul, Prozedur) sollen nur relevante Objekte sichtbar sein. Implementationsdetails werden im Inneren ***verborgen*** und sind nicht zugreifbar.

Was haben wir gelernt!

- **Modell der imperativen Programmierung**
  - Zustand, Zustandsübergang
- **Konzept der Variablen**
  - Wertzuweisung
- **Parameterübergabemechanismen**
  - Wertparameter
  - Variablenparameter
  - Wie geht man damit um
- **Gültigkeit von Bezeichnern**
- **Lebensdauer von Objekten**
- **Konzept der Lokalität**

Glossar

- **Modelle der imperativen Programmierung (von-Neumann-Sprachen)**
- **Programm- und Datenspeicherzustände und -übergänge bei imperativen Programmen**
- **imperative Programmierung: Aufgaben und Hilfsmittel der Programmiersprache**
- **Programmiersprachliche Objekte**
- **Variablenbegriff: typisiert, Initialisierung, LH-Value, RH-Value, Wertzuweisung, Veränderung bei Parameterübergabe**
- **Deklarationen: verschiedene Arten, Zweck dieser Arten**
- **Konstantenbegriff von Modula-3**
- **Prozeduren: Aufgaben, Parameter, Gültigkeit, Lebensdauer, Unterscheidung zu Funktionen**
- **Prozedurdeklaration: Prozedurkopf/Signatur, Prozedurrumpf, Formalparameterliste, lokale Deklarationen, Verwendung der Formalparameter im Rumpf**
- **Prozeduraufruf: Aktualparameterliste, Konsistenz, Aufruf und Deklaration, Semantik durch Inkarnation**
- **Parameter: Arten, Übergabemechanismen, Methodikregeln, Call-by-Value (Aufruf über den Wert), Call-by-Reference (Aufruf über die Adresse)**
- **rekursive Prozeduren (direkt und indirekt): Verwendung bei Divide & Conquer-Entwicklungsstrategie, statisches Verständnis der Rekursion, Laufzeitgeschehen bei rekursiven Prozeduren**
- **Datenaustausch über Prozeduren**
- **Namensraum von Prozeduren, Gültigkeitsbereich, Erweiterung durch Importe, lokale Variable, Vorteile im Sinne der Programmiermethodik**
- **Lebensdauer von Prozedurinkarnationen, darin enthaltener Variablen**