

Einführung in die Programmierumgebung Lazarus

1. Vorbemerkungen

Links:

- Download: <http://www.heise.de/download/lazarus-1159651.html>
- http://wiki.freepascal.org/Lazarus_Tutorial/de
- <http://www.oszhandel.de/gymnasium/faecher/informatik/lazarus/index.html>

Was ist Lazarus?

- Lazarus eine freie Entwicklungsumgebung für Anwendungssoftware
- kann auf unterschiedlichen Betriebssystemen (Windows, Linux) implementiert werden
- Grundlage ist **FreePascal** (eine lizenzfreie kostenlose Programmiersprache)
- hervorgegangen aus dem MEGIDO-Projekt, das 1999 scheiterte
- Lazarus ist eine prozedurale Sprache, die objektorientierte Programmierung vereint. (sh. 6.)

2. Bezeichner

Es werden folgende Regeln zur Wahl von Bezeichnern **festgelegt**, dabei gilt generelle Kleinschreibung! (*Lazaruseigene Bezeichner erkennt man an der Großschreibung.*)

Objekt	Bezeichner-Beispiele
Programmname	a mpel
Projektname	p ampel
Formular	f ampel
Units	u gruen, u rot
Buttons	b druecken; b OK
Editorfelder	e dauer
Label	l ampelfarbe
ListBox	l bxyz
Images	i bild
Shapes (geom. Figur)	s figur
Memofelder	m text

3. Wichtige Hinweise

- **Lege für jedes Lazarus-Projekt einen extra Ordner an!**
- Benenne nie die einmal festgelegten Dateinamen um!
- Öffne nie mehrere Lazarusprojekte in einem Lazarus-Task! (Starte dafür Lazarus mehrfach.)
- Verändere nie die Projektdatei (*.lpr) von Lazarus!
- Änderungen im **interface**-Teil einer **unit** weitestgehend vermeiden – arbeite nur im **implementation**-Teil!
- Nach jeder Änderung im Projekt muss es immer erst kompiliert werden, bevor es gestartet werden kann! (*Fehler werden im Nachrichtenfenster angezeigt*)

4. Dateien eines Lazarus-Projekts

Beim Wählen von „**Speichern unter...**“ müssen zwei Dateien benannt werden:

xxx.pas und **xxx.lpr**

- Es werden immer mehr als 2 Dateien gespeichert. Aber diese beiden sind zu benennen!
- Die Projektdatei (lpr) und die Unitdatei (pas) müssen nicht denselben Namen haben.
- Überblick über alle erzeugten Dateien:

xxx.exe: Das ausführbare Hauptprogramm. *(läuft zum Schluss ohne Lazarus auf jedem Windows 32bit – Rechner)*

xxx.lpi: Dies ist die Hauptdatei eines Lazarus Projekts *(Lazarus Project Information)*

xxx.lpr: Die Hauptprogramm Quelldatei. *(eine völlig normale Pascal Quelldatei; Sie hat eine uses Anweisung, die dem Compiler angibt, welche Units er braucht.)*

xxx.lfm: Hier speichert Lazarus das Layout der Formularunit. *(beschreibt letztendlich das Aussehen des Anwendungsfensters)*

xxx.lrs: Dies ist die erzeugte Ressourcen Datei. *(enthält alle sprach- oder kultur-raumspezifischer Daten)*

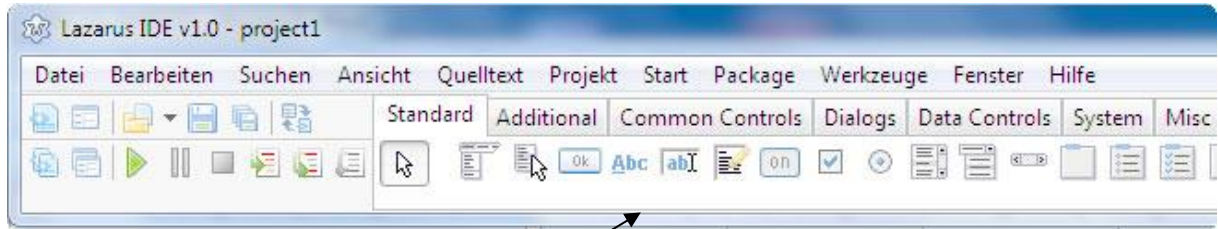
xxx.pas: Die Unit, welche den Programm-Code für das Formular enthält.


xxx.ppu: Dies ist die kompilierte Unit.

ppas.bat: Dies ist ein einfaches Script zum Linken des Programms in eine ausführbare Datei. Wenn die Kompilierung erfolgreich war, wird es vom Compiler gelöscht.

5. Die Lazarus-Oberfläche

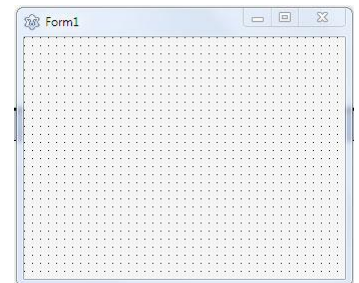
1. Hauptfenster



- dient der **Programmverwaltung**
- enthält die Komponenten-Palette
- wichtig ist der RUN-Button  zum Starten von Programmen

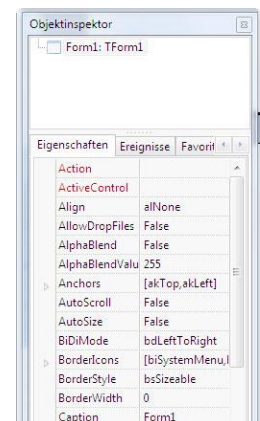
2. Formularfenster

- Das Formularfenster stellt das zentrale visuelle Entwicklungstool von Lazarus dar
- ist **Träger von Komponenten** wie z.B. Button, Eingabefelder, Bezeichnungsfelder, Menüs, ...



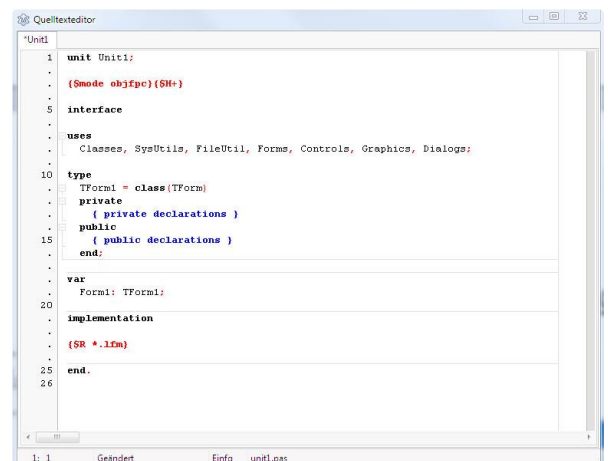
3. Objektinspektor

- Im Objektinspektor können die bereits erwähnten **Eigenschaften des Objektes** (Höhe, Breite usw.) und **Ereignisse** (Verhaltensweisen) eingestellt werden.



4. Editorfenster

- Im Quelltexteditor erfolgt das eigentliche **Programmieren**



6. Arbeit mit Objekten, Klassen und Methoden

Klassen

Klassen stellen den "Bauplan" eines Objekts dar. Sie definieren, welche Eigenschaften (Attribute) und welche Methoden ein Objekt besitzt und wie es auf bestimmte Ereignisse reagiert. Klassennamen beginnen in Lazarus üblicherweise mit einem großen T (für Type).

In Lazarus wird eine Klasse so deklariert:

```
type
  TAuto = class
    private
      Farbe: String;
      Baujahr: Integer;
      Tankinhalt: Integer;
    procedure SetFarbe(Farbe: String);
    public
      procedure Tanken(Liter: Integer);
  end;
```

In Units, zu denen ein Fenster gehört, ist immer bereits die Klasse TForm1 zu finden:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
  public
  end; //Ende der Klasse TForm1

  TAuto = class
    private
      Farbe: String;
      Baujahr: Integer;
      Tankinhalt: Integer;
    procedure SetFarbe(Farbe: String);
    public
      procedure Tanken(Liter: Integer);
  end; //Ende der Klasse TAuto
```

Die Klasse TAuto stellt nun den Bauplan für beliebig viele Instanzen dar. Farbe, Baujahr und Tankinhalt sind Attribute, also Eigenschaften eines TAutos. Außerdem kann eine Klasse Funktionen (*function*) und Prozeduren (*procedure*) enthalten, die etwas mit der Instanz machen, z.B. über das Tanken den Tankinhalt verändern.

Objekte/Instanzen

Instanzen sind nun "lebendige" Objekte, die nach dem Bauplan einer Klasse erstellt wurden, z.B. Formulare, Button, Labels usw. Sie belegen bei der Programmausführung Arbeitsspeicher und können Daten aufnehmen. Von jeder Klasse kann es beliebig viele Instanzen geben, die alle gleich aufgebaut sind, aber unterschiedliche Daten enthalten können.

Um nun eine Instanz von oben beschriebener Klasse TAuto zu erstellen, muss zunächst eine Variable deklariert werden:

```
var MeinAuto: TAuto;
```

Methoden

Methode ist der Oberbegriff für Prozeduren und Funktionen, die Teil einer Klasse sind. Die Methoden stellen das Verhalten eines Objekts dar.

Methoden werden mit ihrem Kopf innerhalb der Klasse deklariert. An einer späteren Stelle im Code folgt die Implementierung der Methode. Diese erfolgt wie bei einer normalen Prozedur oder Funktion, außer dass vor den Methodennamen der Name der Klasse - getrennt durch einen Punkt - geschrieben wird.

Deklaration (im interface- oder implementation-Abschnitt einer Unit):

```
type
  TAuto = class
  private
    Farbe: String;
    Baujahr: Integer;
    Tankinhalt: Integer;
  procedure SetFarbe(Farbe: String);
  public
    procedure Tanken(Liter: Integer);
  end;
```

Implementierung (im implementation-Abschnitt einer Unit):

```
Procedure TAuto.SetFarbe(Farbe: String);
  Begin
    ...
  End;

Procedure TAuto.Tanken(Liter: Integer);
  Begin
    ...
  End;
```

Komponenten in das Formular (TForm) einfügen:

(**Komponenten**= Instanzen/Objekte eines Lazarus-Formulars)

1. Wählen Sie in der Komponentenleiste mit der Maus die gewünschte Komponente durch Doppelklick.
2. Gehen Sie in den Objektinspektor, linke Spalte.
Tragen Sie bei Name den richtigen Komponentennamen ein.
Tragen Sie bei Caption die gewünschte Aufschrift ein.
3. Verschieben Sie die Komponente an die richtige Stelle im Formularfenster.
4. Ziehen an den Anfassecken, um es auf die gewünschte Größe zu bringen.

Manipulation von Instanzen/Objekten

Alle Objekte besitzen Eigenschaften (Attribute), die im Objektinspektor eingestellt **oder** während der Laufzeit durch Methoden manipuliert werden können:

objektname.Eigenschaft (falls keine Wertzuweisung erforderlich)

objektname.Eigenschaft := Wert (mit Wertzuweisung)

7. Aufbau einer Unit

Kopf einer Unit (=Modul) – Beginn der Unit
Kompilerbefehl
der Interface-Teil enthält alle nötigen Definitionen

bestimmt alle notwendigen Module die das Programm
verwendet (z.B. leere Formulare müssen ja auch erst
programmiert werden)

hier wird der Formulartyp festgelegt, **TForm** ist eine
Formularklasse von Lazarus
das Objekt **bklicken** gehört zur Lazarusklasse **TButton**

procedures sind die Methoden zum Verändern der Objekte

private; public bleiben momentan leer

hier wird das Formular benannt; fhelloWorld ist ein Objekt
der Klasse TfhelloWorld

**Hier beginnt der Anweisungsteil, wo die einzelnen
Methoden durch uns definiert werden können.**

Kompilerbefehl, so stehen lassen

erste selbst definierte Methode

zweite selbst definierte Methode

Ende der Unit.

```
unit uworld;  
{$mode objfpc}{$H+}  
interface
```

```
uses  
  Classes, SysUtils, FileUtil, Forms, Controls,  
  Graphics, Dialogs, ExtCtrls, StdCtrls;
```

```
type  
  TfhelloWorld = class(TForm)  
    ltext: TLabel;  
    bklicken: TButton;
```

```
  procedure bklickenClick(Sender: TObject);  
  procedure ltextDbClick(Sender: TObject);
```

```
  private  
    { Private-Deklarationen }  
  public  
    { Public-Deklarationen }  
  end;
```

```
var  
  fhelloWorld: TfhelloWorld;
```

implementation

```
{$R *.lfm}
```

```
procedure TfhelloWorld.bklickenClick(Sender:  
TObject);  
begin  
  ltext.Caption:='Here I am.'  
end;
```

```
procedure TfhelloWorld.ltextDbClick(Sender:  
TObject);  
begin  
  ltext.Caption:= 'Hello World'  
end;
```

```
end.
```

8. Typumwandlungen (Auswahl)

Methode (Funktion)	Umwandlung	Beispiel
IntToStr	GanzZahl in Zeichenkette	eausgabe.text:=IntToStr(zahl)
StrToInt	Zeichenkette in GanzZahl	zahl:=StrToInt(e.ausgabe.text)
FloatToStr	KommaZahl in Zeichenkette	ezinsen.text:=FloatToStr(zinsen)
StrToFloat	Zeichenkette in KommaZahl	zinsen:=StrToFloat(ezinsen.text)
FloatToStrF	KommaZahl in formatierte Zeichenkette	ezinsen.text:=FloatToStrF(zinsen, ffnumber,8,2) ffNumber ... eine Formatvorlage

9. Datentypen (Auswahl)

Typ	Wertebereich	Deklarations-Bsp.	Zuweisung
Byte (nat. Zahlen)	0 .. 255 (nat. Zahlen)	var zahl: byte;	zahl:= 14;
Word (nat. Zahlen)	0 .. 65535	var zahl: word;	zahl:= 1400023;
Integer (ganze Z.)	- 2147483648 bis +2147483647	var zahl: integer;	zahl:= -124556;
Real (Gleitkomma-zahlen)	$\pm(5.0 \times 10^{-324} \dots 1.7 \times 10^{308})$	var zahl: real;	zahl:= 14.56;
Char	1 Zeichen	var zeichen: char;	zeichen:= 'a'
ShortString	255 Zeichen	var zeichen: shortstring;	zeichen:= 'Otto'
String	ca. 2^{31} Zeichen	var zeichen: string;	zeichen:= 'Otto'
Boolean (Wahrheitswerte)	true, false	var richtig: boolean;	richtig:=true;

→ Hinweise zu den Stringtypen:

String= Kette aus beliebigen Zeichen; steht in Hochkommas: 'Otto' , 'U6Z+' , '345'
Variablen vom Typ string können diese Werte speichern: x:= 'Otto', y:='motor'
man kann ZK mit + verknüpfen: z:=x+y → ergibt für z 'Ottomotor'

→ Boolean:

Wahrheitswerte entstehen bei logischen Ausdrücken (Vergleiche mit <,>,<=,>=,=,<>)
ist a:=3, ergibt der Vergleich a>0 den Wert wahr (true);
der Vergleich a=2 aber falsch (false)

Wahrheitswerte kann man auch Variablen (Typ Boolean) zuweisen

Bsp: x:=a>0 → x hat dann den Wert true

Bedingungen bei **If** lassen sich kürzer fassen: **If** x **then**

→ Ordinale Typen

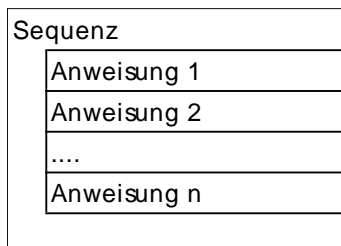
sind Aufzählungstypen; dazu gehören Integer (0,1, ... 9) , Boolean (true, false) ,
Char (a, b, ...z)

11. Programmstrukturen

11.1. Die Sequenz

Die Sequenz besteht immer aus einer Folge von mehreren Anweisungen (Anweisungsfolge), die einmal nacheinander in vorgegebener Reihenfolge abgearbeitet werden. Im einfachsten Fall besteht eine Sequenz aus einer Anweisung.

Struktogramm:



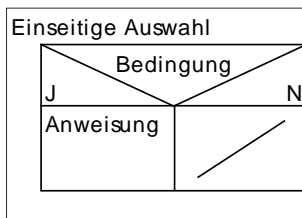
Die Umsetzung im Lazarus-Quelltext erfolgt durch Aneinanderreihung von ein oder mehreren Lazarus-Befehlen, jeweils mit Semikolon getrennt.

11.2. Die Selektion

Die Selektion (Auswahl oder Alternative) ist eine Programmstruktur, die dazu dient, bestimmte Anweisungen nicht auszuführen, wenn eine Bedingung zutrifft.

Man unterscheidet dabei mehrere Arten.

a) Einseitige Auswahl



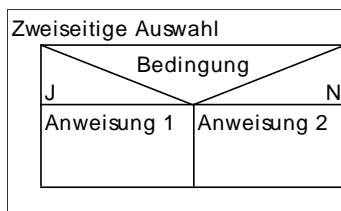
Die *Anweisung* nur wird ausgeführt, wenn die Bedingung zutrifft.

Lazarus:

if (Bedingung) then

begin Anweisung; end;

b) Zweiseitige Auswahl



Die *Anweisung 1* nur wird ausgeführt, wenn die Bedingung zutrifft, sonst wird *Anweisung 2* ausgeführt.

Lazarus:

if (Bedingung) then

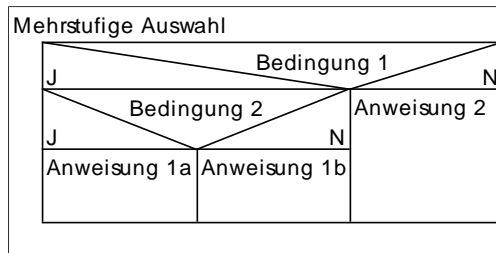
begin Anweisung1; end

else

begin Anweisung 2; end;

Merke: Direkt vor else steht nie ein Semikolon!

c) Mehrstufige Auswahl



Ist eine Verschachtelung von mehreren einseitigen oder zweiseitigen Auswahlstrukturen.

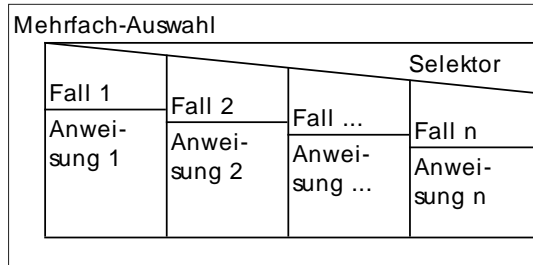
Im nebenstehendem Beispiel wurde die Anweisung 1 durch eine weitere zweiseitige Auswahl ersetzt.

Lazarus:

```

if (Bedingung1) then
    if (Bedingung 2) then
        begin Anweisung1a; end
    else
        begin Anweisung1b; end
else
    begin Anweisung 2; end;
  
```

d) Mehrfachauswahl



In Abhängigkeit von einem Selektor wird sofort der richtige Fall ausgewählt und die entsprechende Anweisung ausgeführt.

Der Selektor ist meist eine ganzzahlige Variable (integer, byte, ..), kann aber auch vom Typ char oder boolean sein.

Lazarus:

```

case selektor of
    wert 1: begin Anweisung1; end;
    wert 2: begin Anweisung2; end;
    wert...: ....
    wert n: begin Anweisung n; end;
end; {of case}
  
```

Merke:

Ist der Selektor ein ganzzahlig Datentyp, dann stehen für wert1, wert2 usw. immer ganze Zahlen (z.B. 1 ; 2 usw.).

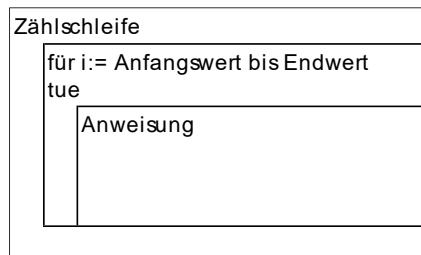
Ist der Selektor vom Typ char, dann stehen für wert1, wert2 usw. immer Zeichen (z.B. 'a' ; 'b' usw.).

11.3. Der Zyklus

Der Zyklus (Wiederholung oder Schleife) ist eine Programmstruktur, die dazu dient, bestimmte Anweisungen mehrfach auszuführen.

Man unterscheidet dabei mehrere Arten.

a) Wiederholung mit fester Anzahl (Zählschleife)



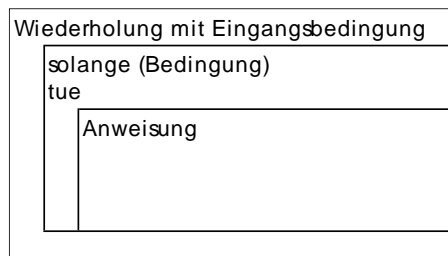
Anweisungen innerhalb einer Zählschleife werden sooft wiederholt, wie vom Anfangswert zum Endwert in Einzelschritten gezählt werden kann. Ist der Anfangswert größer als der Endwert, wird die Zählschleife nicht durchlaufen.

Lazarus: (am Beispiel für Anfangswert 0 und Endwert n)

```
for i:= 0 to n do  
    begin Anweisung; end;
```

b) Bedingte Wiederholung

I) Wiederholung mit Eingangsbedingung (**anfangsgeprüft**)



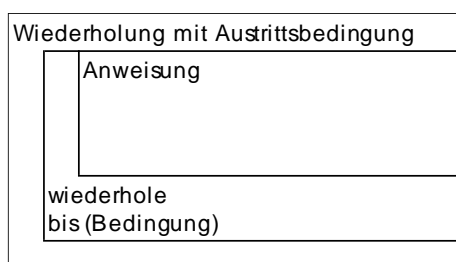
Anweisungen innerhalb einer Wiederholung mit Eingangsbedingung werden solange wiederholt, wie die Bedingung zutrifft.

Merke: Die Bedingung muss sich innerhalb der Wiederholung ändern können, sonst entsteht eine Endlosschleife.

Lazarus:

```
while (Bedingung) do  
    begin Anweisung; end;
```

II) Wiederholung mit Austrittsbedingung (**endgeprüft**)



Anweisungen innerhalb einer Wiederholung mit Austrittsbedingung werden wiederholt, **bis** die Bedingung zutrifft.

Merke: Diese Wiederholung wird immer mindestens einmal durchlaufen, auch wenn die Bedingung bereits

zutreffen sollte. Auch hier muss sich die Bedingung innerhalb der Wiederholung ändern können.

Lazarus:

repeat

 Anweisung;

until (Bedingung);

12. Selbstdefinierte Methoden

Außer Lazarus-Eigene Funktionen (sin, cos, ..) kann man auch eigene Programmroutinen (Methoden) definieren, die innerhalb des Implementation-Teils und sogar innerhalb einer anderen Methode (OnClick, OnActivate, ..) aufgerufen werden können. Man unterscheidet zwischen Funktionen und Prozeduren. Funktionen geben, im Unterschied zu Prozeduren, **immer** einen Wert zurück.

Die Deklaration von Funktionen und Prozeduren erfolgen am Anfang des Implementation-Teils bzw. im Definitionsteil einer Methode – also zwischen **procedure TF....()**; und dem ersten **Begin**.

12.1. selbstdefinierte Funktionen

implementation

....

```
function funktionsname(formale Parameterliste mit Datentyp): Ergebnisdatentyp;  
    {hier evtl. Hilfsvariablen mit VAR ... vereinbaren}  
    begin  
        .....  
        funktionsname:=<Ergebniswert>  
    end;
```

Bsp.:

```
function Kleinere(a,b: integer): integer;  
    VAR h: integer;  
    begin  
        if a<b then h:=a else h:=b;  
        Kleinere:=h;  
    end;
```

Der Aufruf einer Funktions-Methode erfolgt immer in einer Wertzuweisung oder in einer Ausgabeanweisung einschließlich der notwendigen Parameter. Die Reihenfolge der Parameter darf nicht vertauscht werden!

Bsp.: x:=Kleinere(2,5); {hat den Vorteil, dass mit x nun beliebig weitergerechnet werden kann!} **oder**
x:=Kleinere(a,b); wenn a und b konkrete Werte haben; **oder**
Showmessage(inttostr(Kleinere(2,5))); **oder**
Editfeld.Text:= inttostr(Kleinere(2,5);

Die konkreten Werte (hier 2 und 5) heißen jetzt **aktuelle Parameter**.

12.2. selbstdefinierte Prozeduren

Prozeduren sind noch variabler einsetzbar als Funktionen. Prozeduren gibt es ohne oder mit Wertrückgabe, wobei beliebig viele Werte zurückgegeben werden können.

a) ohne Wertrückgabe

implementation

....

```
procedure prozedurname(formale Parameterliste mit Datentyp);  
  {hier evtl. Hilfsvariablen mit VAR ... vereinbaren}  
  begin  
    .....  
  end;
```

Bsp.:

```
procedure KleinereProz(a,b: integer); {a,b sind formale Parameter}  
  VAR h: string;  
  begin  
    if a<b then h:=inttostr(a) else h:=inttostr(b);  
    Showmessage(h+' ist die Kleinere Zahl')  
  end;
```

Der Aufruf einer Prozedur-Methode erfolgt immer nur durch Angabe des Namens einschließlich notwendiger Parameter.

Bsp.: KleinereProz(2,5); {2 und 5 sind jetzt aktuelle Parameter}

a) mit Wertrückgabe

Soll eine Prozedur auch Werte zurückgeben, müssen die Parameter in der Deklaration als Variablen deklariert werden (sgn. Variablenparameter).

Bsp.:

```
procedure KleinereProz2(a,b: integer; VAR h: integer); {a,b,h sind formale Parameter}  
  begin  
    if a<b then h:=a else h:=b;  
  end;
```

Der Aufruf muss genau so viele Werte enthalten, wie in der Deklaration, wobei für den Variablenparameter nur eine Variable möglich ist.

Bsp.: KleinereProz2(2,5,x); **oder**
KleinereProz2(s,t,x); wobei s und t die Zahlenwerte enthalten
x enthält nach dem Prozeduraufruf den kleineren Wert

Die konkreten Werte (hier 2 und 5 und der Wert von x) heißen jetzt **aktuelle Parameter**.

13. Zeichenkettenfunktionen (eine Auswahl)

Für den Datentyp **string** oder **shortstring** gibt es für die Verarbeitung spezielle Methoden. Einige seien hier aufgezählt:

Für die Beispiele seien

s, t, u, v, w Stringvariablen mit den Werten: s:='Otto' t:='motor'; u:='123'; v:='8.75'

x, y Integervariablen mit y:=2

a, b Gleitkommavariablen mit dem Wert a:=2.34

c Variable vom Typ char

a) Length

Liefert die Anzahl der Zeichen einer ZK

Bsp.: x:=length(s) → liefert für x=4

b) StrToInt

Wandelt ZK in ganze Zahl um

Bsp.: x:=StrToInt(u) → liefert für x=123

c) StrToFloat

Wandelt ZK in rationale Zahl (Gleitkomazahl) um

Bsp.: a:=StrToFloat(v) → liefert für a=8.75

d) IntToStr

Wandelt ganze Zahl in ZK um

Bsp.: w:=IntToStr(y) → liefert für w='2'

e) FloatToStr

Wandelt rationale Zahl in ZK um

Bsp.: w:=IntToStr(a) → liefert für w='2.34'

f) AnsiUpperCase

Wandelt String in Großbuchstabe um

Bsp.: w:=AnsiUpperCase(t) → liefert für w='MOTOR'

g) AnsiLowerCase

Wandelt String in Kleinbuchstabe um

Bsp.: w:=AnsiLowerCase(s) → liefert für w='otto'

h) +

Hängt ZK aneinander

w:=s+t → liefert für w='Ottomotor'

w:=t+s → liefert für w='motorOtto'

i) Stringvariable[i]

Betrachtet einen String als Array und greift auf das i-te Zeichen zu

Bsp.: c:= t[3] → liefert das dritte Zeichen von t, also c='t'

14. Arbeit mit externen Dateien

14.1. Arten

a) Textdateien

enthalten druckbare ASCII-Zeichen, die Zeilenweise angeordnet sind
Typvereinbarung erfolgt mit **var dateivariable: *Textfile***;

b) typisierte Dateien

enthalten eine Aufzählung von Daten des gleichen Typs
Typvereinbarung mit: **var Zahlendatei : *File of integer***;
 Mitarbeiter : *File of string*; usw.

14.2. Ablauf der Dateiarbeit

- ① Festlegung, mit welcher Datei gearbeitet werden soll.
- ② Öffnen oder Erstellen der Datei
- ③ Lesen oder Schreiben von Daten aus der/in die Datei
- ④ Schließen der Datei

14.3. Befehle zur Dateiarbeit

zu ①:

AssignFile(Dateivariable, 'Dateiname');
(Dateiname kann ein vollständiger Pfad sein)

zu ②:

Reset(Dateivariable);

öffnet die Datei und setzt den Zeiger auf die erste Zeile (bei Textdateien) bzw. auf das erste Element bei typisierten Dateien

Rewrite(Dateivariable);

erstellt auf dem Datenträger eine Datei; Achtung: Eine vorhandene Datei gleichen Namens wird überschreiben!

zu ③:

Readln(Dateivariable, z);

liest den Inhalt einer Zeile der Textdatei von der aktuellen Zeigerposition und weist ihn der Variablen z zu

Writeln(Dateivariable, z);

schreibt den Wert der Variablen z in die Textdatei an die aktuelle Zeigerposition

→ jeder Lese- oder Schreibvorgang erhöht die Position des Zeigers

→ in typisierten Dateien verwendet man Read bzw. Write!

zu ④:

CloseFile(Dateivariable);

speichert alle Veränderungen und schließt eine Datei

→ Mit Hilfe der Funktion **EOF(Dateivariable)** kann man prüfen, ob das Dateiende erreicht wurde. In diesem Fall wird **true** zurückgegeben, andernfalls **false**.

Nachfolgend ein komplettes Beispiel zum zeilenweisen Lesen aus einer Textdatei:

```

var    beispieldatei: textfile;
        anzahl: integer;
        z: string;

....
begin
    AssignFile(beispieldatei,'quelle.txt');      {bindet Datei an eine Variable}
    Reset(beispieldatei);                        {öffnet Textdatei}
    readln(beispieldatei,anzahl);                {ermittelt die Anzahl der Zeilen}

    if anzahl > 0 then begin
        readln(beispieldatei, z);                {liest die aktuelle Textzeile}
        dec(anzahl);

        .....

    end;

    CloseFile(beispieldatei);                    {schließt die Textdatei}

```