Delphi

Grundlagen

Ein **Delphiprogramm besteht aus 3 Teilen**:
- dem Formular (*.dfm)
- dem Quelltext oder der Unit (*.pas) und
- der Projektdatei (*.dpr), die Quelltext und Formular zusammenfügt.

Änderungen durch den Programmierer sollten nur im Quelltext vorgenommen werden.
Weitere Dateien sind Sicherungsdateien, wenn eine neue Version gespeichert wird.
Sobald ein neues Programm geschrieben wird, soll sofort (!) ein eigenes Verzeichnis angelegt werden und einmal mit
Datei – Speichern unter… die Unit und mit
Datei – Projekt speichern unter… das Projekt gespeichert werden. Später reicht es dann,
Datei – alles speichern aufzurufen.

Ein Delphiprogramm reagiert auf „**Ereignisse**“. Die Art der Reaktion wird im Programmtext festgelegt. Dazu erzeugt Delphi eine **procedure** (auch **Methode** genannt), zwischen **begin** und **end** kann der Programmierer Anweisungen einfügen. Einzelne Anweisungen werden durch ein Semikolon getrennt. Der Beginn einer neuen Zeile im Editor ist unerheblich. Um Anweisungen zu programmieren, sollten zunächst nur Eintragungen im unteren Teil der Unit (Implementation) zwischen die von Delphi erzeugten Schlüsselwörter begin und end geschrieben werden.

Eigenschaften von Formularen, buttons usw. können während der Programmierung im Objektinspektor eingestellt werden. Daneben können sie auch durch das laufende Programm verändert werden, so kann z.B. der Hintergrund des Programms eine andere Farbe bekommen. Um das zu erreichen, benutzt man die Wertzuweisung mit der Zeichenfolge `:=` :
<Eigenschaft> := <Wert>
Beispiel (das Formular heißt Form1):
Form1.Color := clWhite;

Der Programmtext sollte gut lesbar sein, hilfreich sind **Kommentare**. Sie werden von Delphi nicht ausgewertet.
Kurzkommentare, die nur eine Zeile lang gültig sind, gibt man mit „//“ ein:
Form1.Color := clWhite; //Formularfarbe weiß

Längere Kommentare werden durch **geschweifte Klammern**, also zwischen { und }, eingeschlossen.

Ein **Programm** lässt sich durch den Befehl „close“ **beenden**. Dann wird das Programmfenster geschlossen, so als ob man das kleine Kreuz rechts oben im Fenster angeklickt hätte.

Arbeiten mit Variablen

Um Zahlen oder Zeichenketten (Strings) in Delphi zu speichern und zu verarbeiten, benötigt man Variablen. Variablen haben Namen, die keine Leerstellen und keine Sonderzeichen enthalten. Groß- und Kleinschreibung wird nicht beachtet. Ehe man eine Variable benutzen kann, muss sie dem Programm bekannt gemacht werden, man sagt, sie wird „vereinbart“. Dadurch wird Speicherplatz reserviert. Das Vereinbaren geschieht nach der Zeile „`procedure` …“ und vor dem zugehörigen `begin`, eingeleitet durch `var`:

```
procedure TFIdealgewicht.BBerechneClick(Sender: TObject);
var
 gewicht, groesse, idealgewicht, abweichung: real; //lokal
begin
```

Die hier vereinbarten Variablen sind nur innerhalb der Procedure bekannt, sie sind „lokal“ gültig. Sollen Variablen im gesamten Programm zur Verfügung stehen, weil ihre Inhalte in unterschiedlichen Prozeduren benötigt werden, vereinbart man sie hier:

```
implementation
{$R *.dfm}
var steuersatz: real; //global gültig
procedure TFIdealgewicht.BBerechneClick(Sender: TObject);
```

Diese Vereinbarung steht nach „implementation“, aber vor der ersten Prozedur!

Es gibt viele unterschiedliche Variablentypen, wir benötigen zunächst:
real: kann Dezimalzahlen enthalten, intern mit einem Dezimalkomma dargestellt,
integer: enthält nur ganze Zahlen, Datentyp ist „abzählbar“, wird u.a. bei Schleifen zum Zählen der Durchgänge benutzt,
string: Eine Stringvariable kann Texte oder beliebige Zeichenfolgen enthalten, sie müssen mit Hochkomma eingegeben werden: `Adresse:='Horster Straße'`

Wertzuweisungen

Um eine Variable mit einem Wert zu belegen, schreibt man „ := “ (lies: ergibt sich zu). Dabei steht die Variable stets am Anfang. Beispiel für die Variable Nettopreis vom Typ real:

```
Nettopreis:=9.95;
```

Berechnungen

Mit Variablen kann man rechnen und das Ergebnis wieder einer Variablen zuweisen. Beispiel:

```
Endpreis:=Nettopreis*1.19; //19% Mehrwertsteuer mit Wachstumsfaktor 1,19 addiert
Summe:=Summe+2.5; //Inhalt der Variablen Summe wird um 2,5 erhöht und in derselben Variablen
```

gespeichert, dabei wird der alte Inhalt überschrieben.

Bedingungen, Vergleiche

Oft sollen Texte in Abhängigkeit vom Wert einer Variablen ausgegeben werden, z.B. ein Kommentar bei zu hohem Gewicht: Wenn das Gewicht über 100 kg beträgt, dann lautet der Text 'Bitte Diät beginnen!'

```
If gewicht > 100 then Label1.Caption:='Bitte Diät beginnen!';
```

Die Bedingung **`gewicht > 100`** kann erfüllt sein („true“) oder nicht („false“). Auch auf Gleichheit kann man prüfen, hier steht dann das normale „ = “, kein „ := “

```
If gewicht = 100 then …
```

Möchte man größer oder gleich prüfen, schreibt man `>=` (bzw. `<=` bei kleiner oder gleich).

Umwandlungen String → Reelle Zahl und zurück

Alles, was in Edit-Feldern steht, ist zunächst ein String, mit dem nicht gerechnet werden kann. Um Berechnungen durchzuführen, weist man den Inhalt des Editfeldes einer Realvariablen (oder ggf. einer Integervariablen) zu. Das funktioniert aber nicht ohne weiteres, weil es sich um verschiedene Datentypen, eben String bzw. real, handelt. Hier benötigt man die Umwandlungsfunktionen:

```
Preis:=StrToFloat(Edit1.text);
```

StrToFloat ist eine Abkürzung für String to Float, Float ist ein Oberbegriff für Fließkommazahlen wie z.B. real. Wenn man dann ein Ergebnis in einer Realvariablen hat, das wieder ausgegeben werden soll, benutzt man die Funktion FloatToStr (Float to String):

```
Preis:=Preis*1,05;  //Preis wird um 5% erhöht…
Edit2.text:=FloatToStr(Preis); //… und im 2. Editfeld ausgegeben.
```

Besonders bei Divisionen passiert es leicht, dass das Ergebnis mit sehr vielen Nachkommastellen ausgegeben wird:
`Edit3.text:=FloatToStr(5/3)`
erzeugt als Ausgabe 1,66666666666666666.
Um diese Ausgabe vernünftig zu runden, hängt man an FloatToStr noch ein F an und muss dann zusätzlich die Genauigkeit (Anzahl der gültigen Ziffern) und die Anzahl der Nachkommastellen angeben:
`Edit3.text:=FloatToStrF(5/3,ffFixed,10,2)`
ergibt als Ausgabe 1,67 (Genauigkeit 10 Ziffern, 2 Nachkommastellen, gerundet, nicht abgeschnitten).
Noch ein Beispiel:
`edit4.Text:=FloatToStrF(1/7,ffFixed,7,9);`
ergibt 0,142857100. Die Zahl hat zwar 9 Nachkommastellen, wurde aber nur auf 7 gültige Ziffern genau ausgegeben. Falls die Genauigkeit weniger Stellen umfasst als vor dem Komma, wird auf die wissenschaftliche Notation umgeschaltet:
`edit5.Text:=FloatToStrF(1234567,ffFixed,5,2);`
ergibt 1,2346E06, also 1,2346 * 10^6.
Wenn auf diese Weise Variableninhalte ausgegeben werden, bleibt der genaue Wert in der Variablen unangetastet, obwohl die Ausgabe auf 2 Nachkommastellen gerundet ist.

If … then … else

In Delphi sind wie bei den Lego-Robotern solche Bedingungsabfragen möglich. Man schreibt:
`if <Bedingung> then <Anweisung> else <Anweisung>;`
Bedingung und Anweisung muss natürlich mit richtigen Inhalten gefüllt werden. Der else-Teil kann auch wegfallen. Ein Beispiel war oben schon abgedruckt:
`If gewicht > 100 then Label1.Caption:='Bitte Diät beginnen!';`
Sollen mehrere Anweisungen nach dem Schlüsselwort then ausgeführt werden, benötigt man „Klammern", die in Delphi mit `begin` (entspricht öffnender Klammer) und `end` (schließende Klammer) realisiert sind:

```
If gewicht > 100 then
begin
Label1.Caption:='Bitte Diät beginnen!';
Label2.Caption:='Viel Erfolg'
end //then,  erläutert, zu welchem begin das end gehört
else //wichtig: vor else darf kein Semikolon stehen, vor end braucht kein Semikolon zu stehen
Label1.Caption:='Ihr Gewicht liegt im Normbereich!';
```

Im then-Teil stehen zwei Anweisungen, daher ist begin-end nötig. Es ist sinnvoll, durch einen Kommentar hinter dem `end` zu erläutern, wozu es gehört!

Programmentwicklung

Zur Entwicklung eines Programms gehört sorgfältige Planung und anschließend umfangreiches Testen!
Du solltest stets

- **das Formular mit Ein- und Ausgabefeldern sowie Knöpfen (buttons) entwerfen sowie**
- **einen Programmablaufplan entwickeln.**

Beispiel: Programm, das Euro in Dollar umwandelt.
Formular: 2 Editfelder und ein Button zum Berechnen, ggf. ein Button zum Löschen aller Felder
Ablaufplan: Wenn der Button „berechne" gedrückt wird, soll folgendes passieren:
Eurobetrag einlesen und in Variable euro speichern, Umwandlung string → real nicht vergessen!
Euro mit Umrechnungsfaktor multiplizieren und in Variable Dollar speichern,
Variable Dollar in zweitem Editfeld ausgeben, dabei real → String umwandeln!
Mögliche Erweiterung:

- Der aktuelle Kurs (=Umrechnungsfaktor, steht z.B. bei Spiegel.de) kann eingegeben werden.

- Der aktuelle Kurs wird in eine neue Variable geschrieben und mit dem vorherigen Verglichen, als Kommentar wird ausgegeben: Der Kurs des Dollar gegenüber dem Euro ist gestiegen (bzw. gefallen).

Bei umfangreichen Programmen ist es sinnvoll, den Ablaufplan mit ein paar Werten zu testen und die Berechnungen mit dem Taschenrechner durchzuführen. Dabei kann man logische Fehler entdecken und berichtigen.

Wiederholungen

Es gibt 3 wichtige Typen von Schleifen:

1. Die Zählschleife (for…to…do-Schleife), bei der man die genaue Anzahl der Durchläufe kennt,
2. die annehmende Schleife, die mindestens einmal durchlaufen wird (repeat-Schleife),
3. die abweisende Schleife, die vor dem ersten Durchgang eine Eingangsbedingung prüft (while-Schleife).

Die Zählschleife (For-Schleife)

Hier wird eine (lokale) Variable als Schleifenzähler benötigt, die „abzählbar“ ist, meist vom Typ integer.
Beispiel:

```
procedure summe(n); //berechnet die Summe der ersten n Zahlen
var  i: integer;
     summe:real; //damit kein Überlauf wegen großer Zahlen entsteht
begin
summe:=0;
for i:=1 to n do summe:=summe+i;
end;
```

Sollen mehrere Befehle in der Schleife ausgeführt werden, so sind sie in begin – end einzuschließen. Die Zählschleife kann auch rückwärts zählen, dann wird `to` durch `downto` ersetzt: `for i:=n downto 1 do` …

Die Repeat-Schleife

Diese Schleife wird mindestens einmal durchlaufen. Am Ende wird eine Bedingung geprüft. Ist sie wahr, so wird die Schleife beendet, anderenfalls geht es von vorn (bei repeat) erneut los.
Repeat und until wirken dabei wie eine Klammer, so dass auch mehrere Befehle in der Schleife ohne begin-end stehen können. Wichtig: In der Schleife muss die Bedingung so verändert werden, dass sie irgendwann mal wahr wird, sonst steckt das Programm in einer Endlosschleife fest.
Beispiel:

```
procedure anzahl(Obergrenze);
{berechnet, wie viel Zahlen von 1 bis n aufsummiert werden müssen, um eine
bestimmte Zahl zu erreichen}
var  i: integer; summe: real;
begin
summe:=0;i:=1; //Startwerte setzen
repeat
summe:=summe+i;
i:=i+1
until summe>= Obergrenze; //vor until kein Semikolon!
end;
```

Die While-Schleife

Hier wird gleich am Anfang eine Bedingung geprüft. Ist sie wahr, so wird die Schleife durchlaufen, anderenfalls geht es danach weiter. Wenn mehrere Befehle zur Schleife gehören sollen, sind sie mit begin-end zu klammern.
Wichtig: In der Schleife sollte die Bedingung so verändert werden, dass sie irgendwann mal falsch wird, sonst steckt das Programm in einer Endlosschleife fest.
Beispiel der repeat-Schleife jetzt mit einer while-Schleife:

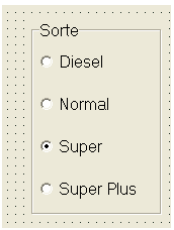
```
procedure anzahl(Obergrenze);
{berechnet, wie viel Zahlen von 1 bis n aufsummiert werden müssen, um eine
bestimmte Zahl zu erreichen}
var  i: integer; summe: real;
begin
summe:=0;i:=1; //Startwerte setzen
while summe<Obergrenze do
begin
 summe:=summe+i;
 i:=i+1;
end; //der while-Schleife
end; //der Prozedur
```

Wichtige Ergänzung zu den Schleifen:
Wird eine Schleife abgearbeitet, so reagiert das Programm nicht auf Änderungen an der Programmoberfläche. Insbesondere ist es nicht ohne weiteres möglich, den Schleifendurchlauf durch einen Stop-Button zu beenden.
Damit das trotzdem funktioniert, kann man die Anweisung

```
Application.ProcessMessages;
```

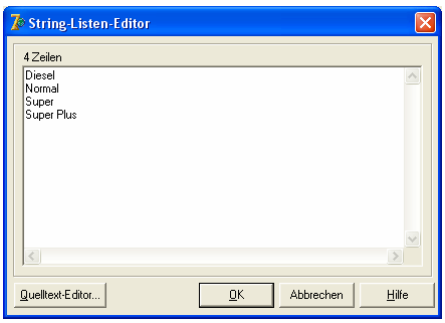
in den Schleifenkörper integrieren.
Sie sorgt dafür, dass das Programm kurz anhält und nachsieht, ob sich an der Oberfläche etwas verändert hat, etwa ein Button gedrückt wurde. Dadurch kann man mit einem Knopf z.B. eine boolesche Variable „fertig“ (Datentyp boolean, kann „true“ oder „false“ sein) mit „true“ belegen und auf diese Bedingung am Schleifenausgang prüfen.

Radiobuttons, Mehrfachauswahl mit Case



Radiobuttons erinnern an alte Radios, wo man Tasten für UKW, Mittelwelle, Langwelle hatte. Drückte man UKW, so sprang die vorher gewählte Taste heraus. Hier ist also genau eine Auswahl aus mehreren Alternativen möglich.
Am einfachsten ist es, eine „Radiogroup“ zu verwenden (Leiste Standard, 3. Symbol von rechts).

Die Anzahl der Knöpfe legt man in der Eigenschaft „items“ fest (auf … klicken, in der sich öffnenden Box entspricht jede Zeile einem Button):



Welcher Button bei Programmstart ausgewählt ist, ergibt sich aus der Eigenschaft ItemIndex.
-1 bedeutet keiner, Zählung beginnt bei 0, also 0 = erster Button!!!
Oben im Beispiel ist 0 = Diesel, 1 = Normal, 2 = Super und 3 = Super Plus.

Heißt die Radiogroup RSorte, dann kann man mit folgendem Code über die Eigenschaft ItemIndex den aktuellen Literpreis (hier: Sortenpreis) ermitteln. Die Preise werden dann aus dem entsprechenden Edit-Feld entnommen.

```
case RSorte.Itemindex of
0: Sortenpreis:=StrToFloat(EDiesel.Text);
1: Sortenpreis:=StrToFloat(ENormal.Text);
2: Sortenpreis:=StrToFloat(ESuper.Text);
3: Sortenpreis:=StrToFloat(EPlus.Text);
end; //case
```

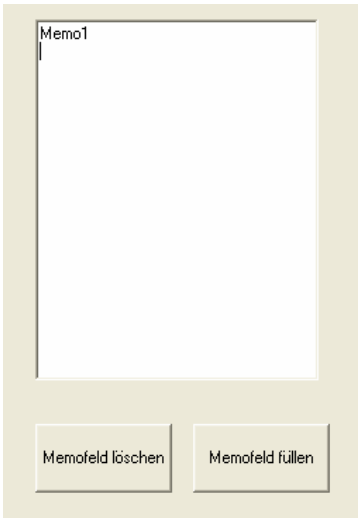
Mit der case-Anweisung spart man sich viele if-Anweisungen. Sie funktioniert bei abzählbaren Variablen wie z.B. integer.

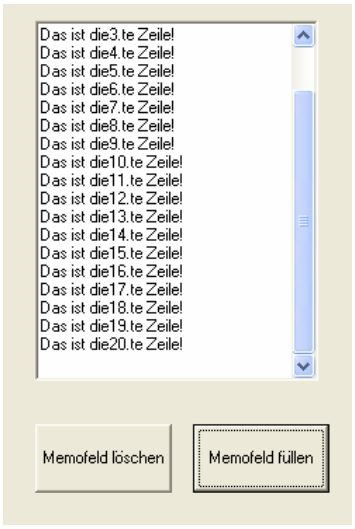
Syntax:

```
case <Integervariable> of
0: <Anweisung>;
1: begin < Anweisung; Anweisung; Anweisung > end;
2: < Anweisung >;
3..5: <Anweisung>; //auch mehrere Zahlen in einer Reihe möglich
end; //nötig, um die Mehrfachauswahl mit case zu beenden.
```

Memofelder

Im Gegensatz zu Edit-Feldern können Memofelder auch mehrzeiligen Text aufnehmen.
Memofeld löschen: `Memo1.Clear;`
Memofeld füllen: `Memo1.Lines.Add(<string>);`
Auf einzelne Zeilen zugreifen:
Die Variable “Ort” sei vom Typ string, dann schreibt man z.B.
Ort:=Memo1.Lines[2]; //die 3. Zeile des Memofeldes (Zählung beginnt bei Null) wird „Ort“ zugewiesen.
Mit der Eigenschaft „scrollbars“ kann man waagerechte und senkrechte „Schieber“ erzeugen, die man benutzt, wenn der Inhalt eines Memofeldes nicht in das Fenster hineinpasst.
Memofeld ohne Scrollbars (ssNone): Memofeld mit vertikaler Scrollbar (ssVertikal):

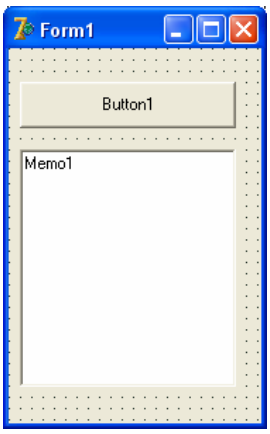




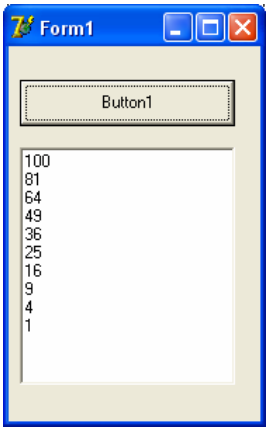
Arrays

Arrays (Felder) sind besondere Variablen. Sie können nicht nur einen Inhalt aufnehmen, sondern sind wie ein Schrank mit durchnummerierten Schubladen. Auf die einzelnen Schubladen kann man mit einer Indexzahl, die in eckigen Klammern angegeben wird, zugreifen. Am einfachsten wird das an einem Beispiel deutlich:

Das folgende Programm schreibt die Quadratzahlen von 1 bis 10 in das Array Zahlen. Danach werden die Daten rückwärts, also 10^2, 9^2, …, 1^2 in das Memofeld geschrieben.




```
implementation

{$R *.dfm}

var zahlen: array[1..10] of integer;

procedure TForm1.Button1Click(Sender: TObject);
var i: integer;
begin
for i:=1 to 10 do zahlen[i]:=i*i; //Quadratzahlen im Array
speichern
memo1.clear;
for i:=10 downto 1 do memo1.lines.Add(IntToStr(zahlen[i]));
//Array rückwärts ausgeben

end;

end.
```



Im Detail:
`var zahlen: array[1..10] of integer;`
zahlen ist der Name des Arrays, [1..10] gibt die Anzahl der „Schubladen“ an, integer legt fest, welche Art von Daten in die einzelnen Schubladen passen.
`var Namen: array[1..1000] of string;`
Hier können 1000 Namen gespeichert werden. Um den Namen in der Schublade 512 in die Stringvariable Nachname zu übernehmen, schreibt man
`Nachname:=Namen[512];`

Arrays erlauben es, große Datenmengen zu speichern und gezielt auf einzelne Elemente mit der Indexzahl in eckigen Klammern zuzugreifen, zugleich kann man auch über einen Schleifenzähler i wie im Beispiel oben mit einer Schleife alle Schubladeninhalte ausgeben, z.B. in ein Memofeld.

Aufgabe:
Schreibe ein Programm, das 20 Zufallszahlen (integer) aus dem Bereich von 1 bis 1000 in ein Array mit dem Namen **Zufallszahlen** schreibt und anschließend in 2 Labels ausgibt, welches die größte (max) und welches die kleinste Zahl (min) ist.
Tipp: Weise die erste der erzeugten Zufallszahlen, also Zufallszahlen[1] den Variablen min und max zu. Prüfe dann in einer Schleife, ob die jeweils aktuelle Zufallszahl <min ist und ob sie >max ist. Wenn ja, wird die aktuelle Zufallszahl in min bzw. max gespeichert und überschreibt die alte.
Beispiel mit 5 Zahlen: Erzeugt wurden die Zufallszahlen 17; 13; 28; 93; 5
Wir belegen die Variablen min und max mit der ersten Zufallszahl, hier also 17.

Schleifenzähler i	Zufallszahl[i]	min	max
	17	17	17
2	13	13	17
3	28	13	28
4	93	13	93
5	5	5	93

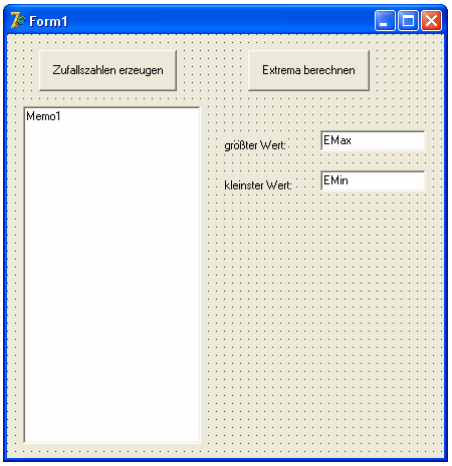
Jetzt stehen in min die kleinste und in max die größte der erzeugten Zufallszahlen, also 5 bzw. 93.

Lösung der Aufgabe oben:

Unit1.pas

```
implementation

{$R *.dfm}

var Zufallszahlen: array[1..20] of integer;

procedure TForm1.BErzeugenClick(Sender: TObject);

var i: integer; //lokale Laufvariable innerhalb procedure

begin
randomize; //Zufallszahlengenerator initialisieren
memo1.Clear; EMax.Text:=''; EMin.Text:=''; //Formular löschen
//Schubladen füllen und in Memo1 anzeigen
for i:=1 to 20 do
  begin //for
  Zufallszahlen[i]:=random(1000)+1; //Zufallszahl zwischen 1 und 1000
  memo1.lines.add(IntToStr(Zufallszahlen[i]));
  end; //for

end;

procedure TForm1.BExtremaFindenClick(Sender: TObject);
var i, max, min: integer;
begin
max:=Zufallszahlen[1]; min:=Zufallszahlen[1];
for i:=2 to 20 do
 begin //for
 if Zufallszahlen[i]<min then min:=Zufallszahlen[i];
 if Zufallszahlen[i]>max then max:=Zufallszahlen[i];
 end; //for
EMax.Text:=IntToStr(max);
EMin.Text:=IntToStr(min);
end;

end.
```

64: 5 Verändert Einfügen



Ergebnis eines Programmlaufs: