Duale Hochschule Baden-Württemberg
Stuttgart
University of
Cooperative Education

Programmieren in Pascal und Delphi

Grafische Anwendungen

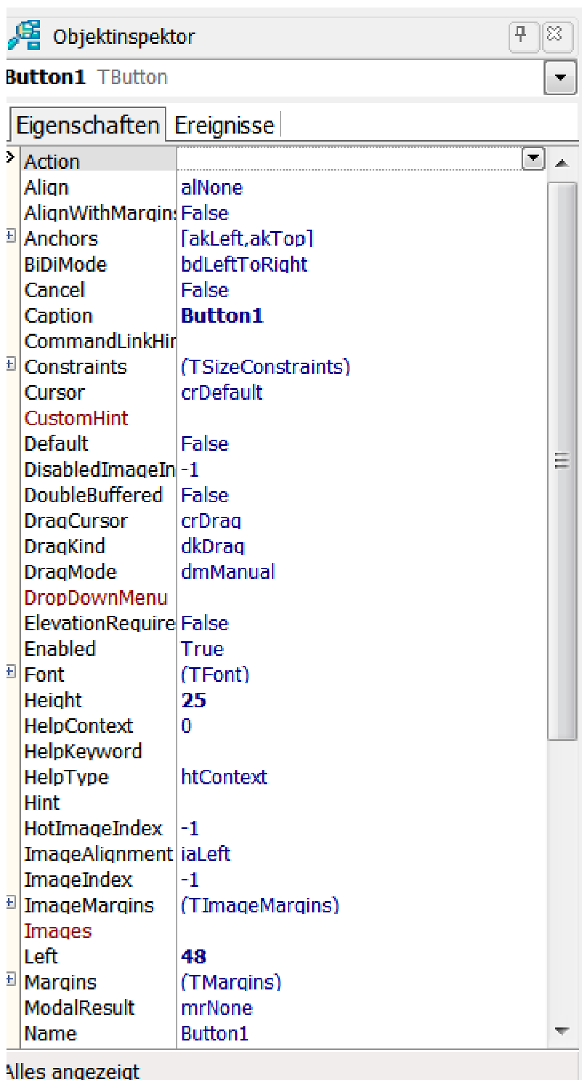
Vorarbeit: Klicken sie, bevor sie irgend etwas machen ins Datei Menü unter „Alles Speichern“ und wählen sie „Eigene Dateien“ als Basis und nicht ein Netzlaufwerk – sie können auf dieses nicht compilieren.

Anders als die Konsolenprogrammen bestehen grafische Anwendungen aus mehreren Dateien. Es sind deswegen beim ersten Compilieren mehrere Speichervorgänge nötig. Brechen sie nicht nach dem ersten ab und geben sie nicht der zweiten datei den gleichen Namen!

Jede Grafische Anwendung besteht aus mindestens zwei Teilen:

- Einer Projektdatei (Endung .dpr) – das Hauptprogramm das alle Teile einbindet
- Mindestens einem Formular. Dieses wiederum besteht aus einer Codedatei (Endung .pas) und einer Beschreibungsdatei des grafischen Aufbaus (Endung .dfm). Jedes Formular muss einen eigenen Namen tragen.
- Pro Formular wird ein neues Paar von Dateien erstellt.

Wenn sie irgendwann einmal ein Fenster nicht mehr sehen, so können sie es im Ansicht Menü wiederherstellen. Die beiden wichtigsten (Designansicht des Formulars) und das Codefenster erhalten sie mit F12 und F11.



Nach einer Änderung müssen sie das Programm neu übersetzen (F9). Änderungen im Code werden nicht automatisch in das laufende Programm übernommen. Dieses beenden sie wie jedes andere Windows Programm durch einen Klick auf das „x“ oben rechts.

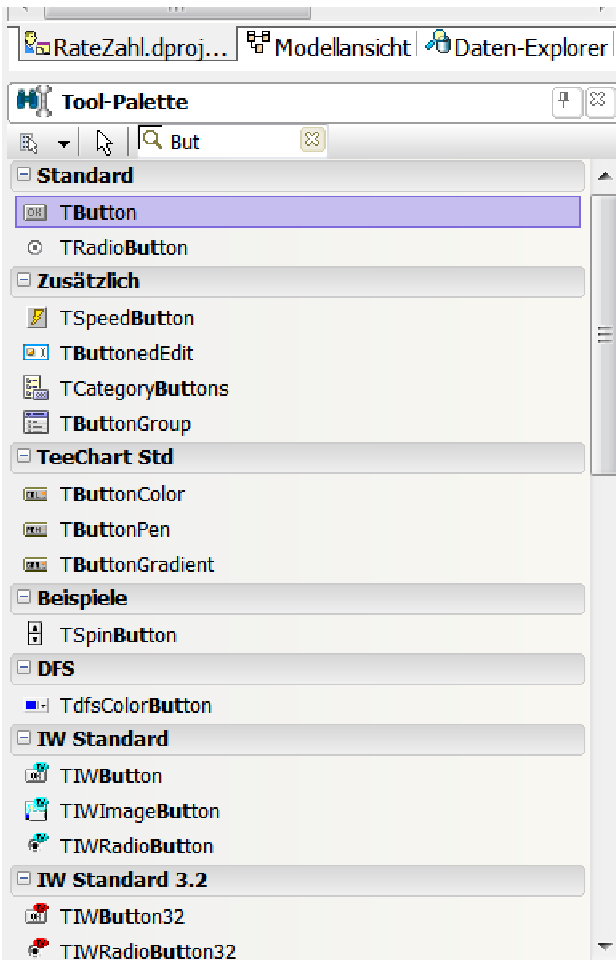
Heute lernen wir die Oberfläche von Delphi kennen. Eine neue grafische Anwendung wird erstellt indem man im Menü bei **Neu → Neue VCL Formularanwendung** klicken. Es öffnet sich ein Fenster in der Entwurfsansicht. Dieses Fenster hat Eigenschaften, die wir verändern können. Die bequemste Möglichkeit dazu ist der **Objektinspektor** (Bild links).

Er hat zwei Reiter: einen mit den Eigenschaften und einen mit den Ereignissen. Die wichtigste Eigenschaft jeder Komponente ist ihr **Name** Sie sollten den Namen nur über den Objektinspektor ändern und nicht wie bisher, indem sie im Quelltext diesen ändern.

Sie sehen, dass ein einfaches Fenster schon eine Menge von Eigenschaften hat. Im Folgenden werden wir einige im Objektinspektor ändern.

Wenn sie einen neuen String bei „**Caption**“ eingeben, so ändern sie die Beschriftung der Titelleiste. Viele Elemente die eine Beschriftung haben, verfügen über diese Eigenschaft.

Den Font kann man in **Font** einstellen, ebenso die Farbe der Beschriftung. Die Hintergrundfarbe kann mit **Color** verändert werden und **Width** und **Height** geben die



Abmessungen des Fensters an. Die Größe kann auch durch Verändern des Fensters in der Designansicht verändert werden.

Verändern sie einige dieser Eigenschaften, compilieren sie die Anwendung und sehen sie wie sich ihre Änderungen auswirken.

Wir können die Eigenschaften aber auch im Programm verändern. Das hat den Vorteil, dass es zur Laufzeit erfolgt und nicht einmal festgelegt ist. Dazu eine kleine Demonstration.

Fügen sie zu dem Fenster (Fachausdruck: **Formular**) zwei **Buttons** hinzu. Die Buttons finden sie in der **Komponentenliste** unten Links (siehe Abbildung). Da es über 400 Komponenten gibt (die meisten sind grafische Elemente) ist es am einfachsten eine zu finden, indem man in die Suchbox einen Text eingibt. So findet man einen Button durch den Text „But“ schnell. Sie stellen auch fest, es gibt davon ziemlich viele. Wir brauchen den Standardbutton „Tbutton“.

Wiederholen sie es mit einem zweiten Button. Sie haben nun zwei Buttons mit der Beschriftung „Button1“ und „Button2“.

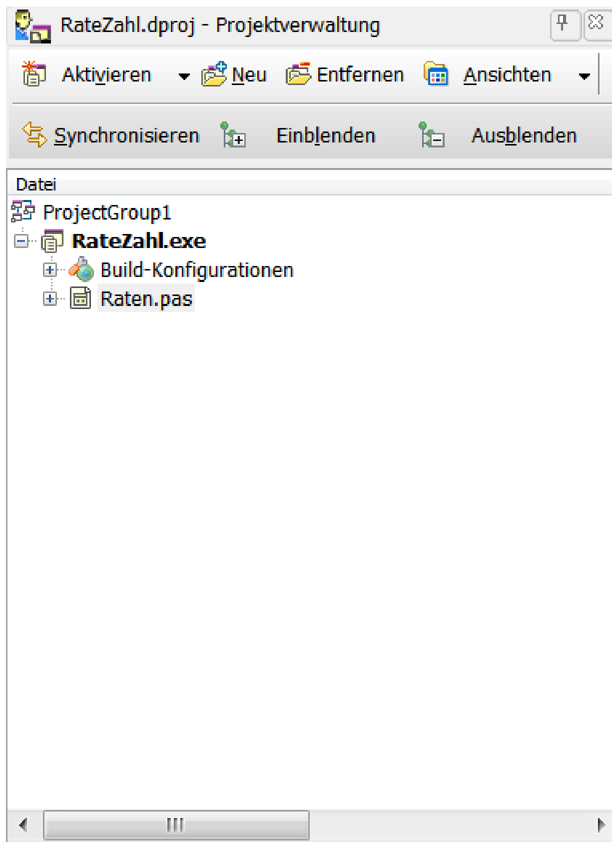
Ändern sie nun im Objektinspektor für jedes Element die Eigenschaft Caption auf „Kleiner“ und „Größer für die beiden Buttons.
Nun wollen wir einmal selbst Code schreiben. Dabei gibt es einen wichtigen Unterschied zu den Konsolenprogrammen: Wir schreiben nun ereignisorientiert. Das Programm ruft bei einem Klick auf den Button eine Prozedur auf und wir können Code in den Block einfügen, der ausgeführt wird, wenn der Anwender auf die Prozedur klickt. Weiterhin legt nun das System den Rumpf der Prozedur fest, da diese eine vorgegebene Parameterliste hat (den Namen können sie selbst festlegen). Es ist wichtig zu vergegenwärtigen, dass nun am Code Sie und das System arbeiten. Wenn sie wichtige Teile des Programmes einfach im Code umbenennen, löschen oder sonst wie verändern, kann ihr Code nicht mehr compiliert werden. So verändern sie den Code:

- Sie wollen eine Methode oder Komponente umbenennen: Tun sie dies im Objektinspektor, indem sie bei Name (Komponenteneigenschaften) einen neuen Namen eintippen. Methoden können sie umbenennen, indem sie im Objektinspektor den Namen ändern.

- Sie wollen eine Komponente löschen: Markieren sie sie im Designer und drücken auf „Entf“. Die Methode bleibt im Code wird aber wird nicht mehr aufgerufen,

- Sie wollen eine Methode löschen: Gehen sie in den Code und löschen sie alles zwischen begin und end; (aber nicht das Begin und End!) Nach dem nächsten Compilieren wird die Methode gelöscht.

Markieren sie dazu den ersten Button, den sie mit „Kleiner“ beschriftet haben. Wechseln sie in den Reiter „**Ereignisse**“. Dort finden sie ein Ereignis namens „**OnClick**“. Klicken sie in den Bereich rechts daneben.



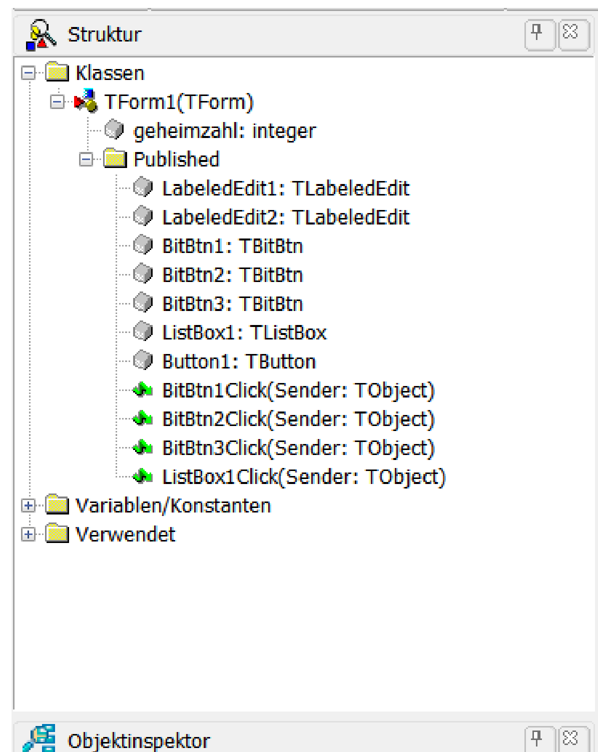
Es öffnet sich nun das Codefenster und sie finden folgenden Quelltext vor:

```
procedure TForm1.Button1Click(Sender:
TObject);
begin

end;
```

Dies bedeutet Folgendes: Sie schreiben Code für das Ereignis „OnClick“ für den Button1 im Fenster 1 (Tform1). Dieses Ereignis ist mit diesem Button verknüpft und diese Prozedur wird immer aufgerufen, wenn sie auf den Button klicken. Schreiben sie nun Code in diese Prozedur und das entsprechende OnClick Ereignis für den zweiten Button:

```
procedure TForm1.Button1Click(Sender:
TObject);
begin
  width:=width-(width div 10);
end;

procedure TForm1.Button2Click(Sender:
TObject);
begin
  width:=width+(width div 10);
end;
```

Dieser Code vergrößert und verkleinert die Breite des Fensters. Da **Width** eine Integer Variable ist, muss mit „**div**“ eine Ganzzahldivision durchgeführt werden (/ als Operator für eine normale Division kann Fließkommazahlen erzeugen und daher nicht verwendet werden).



Lassen sie das Programm laufen und klicken sie auf die Buttons. Was passiert? Sie sehen: Wir können alle Eigenschaften auch durch das Programm verändern. Wichtig könnte z.B. das Ändern von „**Caption**“ sein, um dem Anwender Statusinformationen zukommen zu lassen.

Die Eigenschaft „**OnClick**“ ist eine der wichtigsten. Nahezu alle Komponenten haben sie. Als **Komponenten** bezeichnet man die in der Palette unten befindlichen grafischen Elemente wie Buttons, Editfelder, Labels aber auch Dialoge.

Das Fenster rechts ist die Projektverwaltung. Sie erlaubt es, auf die einzelnen Dateien eines Projektes schnell zuzugreifen. Wenn sie aus Versehen ein neues Formular, anstatt ein neues Projekt erzeugt haben, finden sie dort oft mehrere Dateien. Sie können auch mehrere Projekte zu einer Gruppe kombinieren.

Das Fenster oben links gibt die Struktur des Projektes

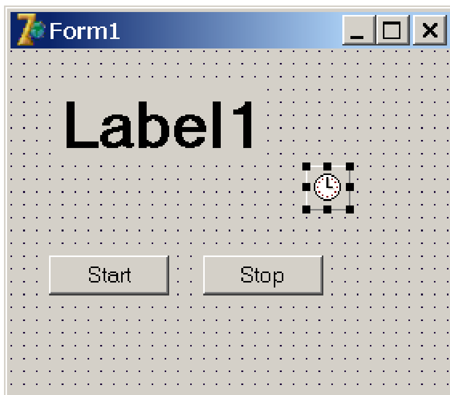
wieder und erlaubt es ein Element schnell anzuspringen oder bei grafischen Elementen auch zu markieren. So sieht das Fenster aus, wenn die Codeansicht aktiv ist:

Wenn sie eine Prozedur (grün) anklicken, so befinden sie sich im Code bei dieser. Wenn sie eine Komponente (graue Box) anklicken, so springen sie zu deren Definition.

Nun wollen wir unsere erste Anwendung schreiben. Dies soll zuerst eine Stoppuhr, dann eine Uhr werden. Dazu fügen wir in ein leeres Formular folgendes ein:

- Zwei **Buttons** (Reiter „Standard“)
- Ein **Label** (Reiter „Standard“)
- und einen **Timer** (Reiter „System“)

Beschriften sie die Buttons und vergrößern sie die **Font** Eigenschaft des **Labels**. Die Anwendung sollte nun so aussehen:



Wir wollen nun eine Uhr bauen. Dazu brauchen wir den **Timer**. Der Timer ist eine nicht visuelle Komponente. Das bedeutet, man sieht sie bei der Laufzeit (wenn das Programm ausgeführt wird) nicht. Aber sie hat eine einfache Funktionalität. Ein Timer hat zwei Eigenschaften. Die Eigenschaft „**enabled**“ schaltet ihn an und aus. Die Eigenschaft „**Interval**“ gibt ein Zeitintervall an, in dem der Timer aufgerufen wird. Der Standard Eintrag „1000“ steht für einen Aufruf alle 1000 ms, also einmal pro Sekunde.

Ist der Timer eingeschaltet (**enabled=true**) so springt er nach Ablauf des Intervalls die Prozedur „**OnTimer**“ an. Diese ist natürlich zuerst mal noch leer. Doch zuerst einmal wenden wir uns den Buttons zu. Der Erste soll die aktuelle Uhrzeit stoppen, der zweite die Differenz ausgeben.

Also nehmen wir bei dem **OnClick** Ereignis die Uhrzeit und ziehen sie beim Click auf den zweiten Button von der aktuellen Zeit ab.

```
procedure TForm1.Button1Click(Sender: TObject);

begin
  uhrzeit := Now;
end;

procedure TForm1.Button2Click(Sender: TObject);

begin
  Label1.caption:= TimeToStr(Now-Uhrzeit);
end;
```

Label1 ist der Name des Labels. Das wir zur Ausgabe der Uhrzeit verwenden. Die Variable Uhrzeit müssen wir noch deklarieren. Dies geschieht am besten in der „private“ Sektion des Formulars. Diese fügen wir unter dem Schlüsselwort „**private**“ ein:

```
  private
    { Private-Deklarationen }
    Uhrzeit : Tdatetime;
  public
    { Public-Deklarationen }
```

Now ist eine Systemfunktion, welche die aktuelle Uhrzeit liefert. Diese wird mit **timetostr()** in einen String umgewandelt. Standardmäßig ist der Timer beim Start aktiv. Von Nachteil ist dass die Uhrzeit nun maximal Sekunden zeigt. Das ist zwar geeignet für längere Tätigkeiten, entspricht aber nicht dem Verhalten einer normalen Stoppuhr, die auch Sekundenbruchteile anzeigt. Dazu ändern wir die Zeile beim Stoppbutton wie folgt ab:

```
  label1.caption:=formatdatetime('ss.zzz',now-uhrzeit);
```

Damit bestimmen wir die Formatierung ein „s“ im String steht für die Sekunden und ein „z“ für die Ausgabe der Millisekunden, also hier: Zwei Stellen für die Sekunden (mit führender Null), drei für die Millisekunden.

Damit haben wir eine Stoppuhr. Aber es fehlt noch die Anzeige der aktuellen Uhrzeit. Dies kann entweder in einem zweiten Label erfolgen (fügen sie eines mit der Bezeichnung „Label2“ ein, oder in der Titelleiste des Formulars. Suchen sie sich eine Lösung aus.

Gehen sie dazu in das „OnTimer“ Ereignis von Timer1. Und fügen sie folgende Zeile ein:

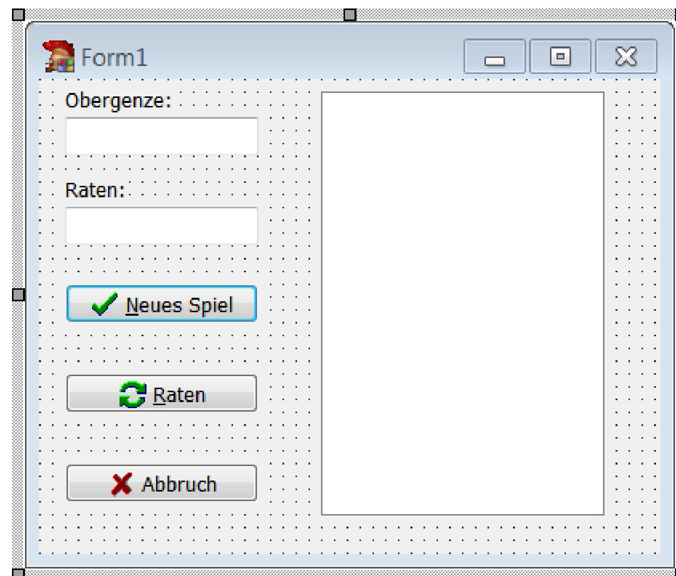
```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  label12Caption:=timetostr(now); // wenn sie ien Label benutzen
  Caption:=timetostr(now); // wenn sie die Titelleiste bevorzugen
end;
```

Beim Formular ist nicht der Name angebbar, wenn man auf ein Feld zugreifen will, es ist der Container der alle eingefügten Komponenten enthält. Standardmäßig ist der Timer beim Start aktiv. Das Intervall für den Aufruf der OnTimer Prozedur kann im Objektinspektor geändert werden.

Das Zahlenratespiel

Das zweite Programm soll ein kleines Ratespiel ergeben. Die Aufgabe: Der Computer sucht sich eine Zahl aus und sie müssen sie in möglichst wenigen Versuchen erraten.

Unser Formular sieht zuerst mal so aus:



Obergenze und Untergrenze sind vom Typ **TlabeledEdit**. Die Buttons vom Typ **TbitBtn**. Die grafische Darstellung wird durch die Eigenschaft **Kind** eingestellt. Ein eigenes Bild kann mit **Glyph** gewählt werden und die Beschriftung des Labels beim Tlabeledit steht in **Editlabel** und dort unter **Caption**.

Rechts steht ein leeres Listenfeld (**Tlistbox**). Beim Klicken auf „Neues Spiel" wird ein neues Spiel gestartet. Die in der „Obergrenze" eingetippte Zahl ist die Obergrenze für eine zu ratende Zufallszahl. Danach wird das Eingabefeld ausgegraut. Das geschieht mit folgendem Code:

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  randomize;
  try
    geheimzahl  := Random(StrToInt(LabeledEdit1.Text))+1;
    LabeledEdit1.enabled:=false;
    listbox1.items.clear;
  except
    Messagedlg(LabeledEdit1.text+' ist keine ganze Zahl!',mterror,[mbok],0);
    Exit;
  end;
end;
```

Dazu muss in der „**private**" Sektion eine neue Variable namens „**geheimzahl**" eingefügt werden. Variablen, die von einer grafischen Anwendung benutzt werden, sollten nach den Schlüsselwörtern „**private**" und „**public**" deklariert werden. Das Schlüsselwort „Var" ist dazu nicht notwendig.

Der Code macht folgendes:

In einem **try .. except** block wird versucht. die Eingabe in eine Zahl zu verwandeln. Gelingt dies nicht, so wird die Funktion verlassen. Vorher wird eine Fehlermeldung ausgegeben. Die genaue Syntax der **Messagebox** nehmen wir noch durch. An dieser Stelle soll nur soviel gesagt werden, dass die dargestellte Box eine Windows-Fehlermeldungsbox ist, mit dem Text und einem Button, der mit "OK" beschriftet ist.

Gelingt die Konvertierung, so wird das Eingabefeld ausgegraut. Es ist nun keine Eingabe mehr möglich (**Enabled**:=false) zudem wird der Inhalt der Listbox gelöscht.

Der Butrton für Abbrechen schaltet die Eingabe wieder ein:

```
procedure TForm1.BitBtn3Click(Sender: TObject);
begin
```

```
  LabeledEdit1.enabled:=true;
end;
```

Der letzte Button für Raten nimmt folgenden Code auf:

```
var zahl : integer;

begin
  try
    zahl := Strtoint(labelededit2.Text);
  except
    Messagedlg(LabeledEdit2.text+' ist keine ganze Zahl!',mterror,[mbok],0);
    exit;
  end;
  if geheimzahl=zahl then
  begin
    Messagedlg('Richtig geraten!',mtinformation,[mbyes],0);
    BitBtn3Click(sender);
    exit;
  end
  else
  if zahl>geheimzahl then
  Listbox1.items.Add('Geraten : '+labelededit2.Text+' - ist größer als die geheime
Zahl')
  else
  Listbox1.items.Add('Geraten : '+labelededit2.Text+' - ist kleiner als die geheime
Zahl');
end;
```

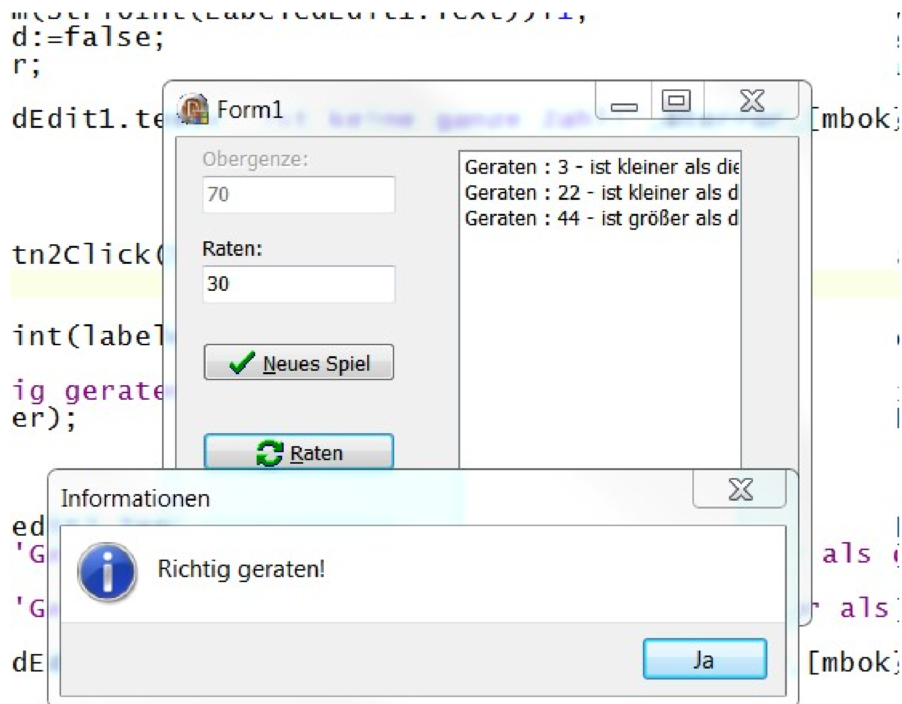
Die Eingaben führen jeweils zu einem weiteren Eintrag in der Listbox, der uns über die bisherigen Rateversuche informiert. Damit arbeitet das Programm. Es kann noch notwendig sein die Listbox zu verbreitern, doch das stellt man nach einem Testlauf fest.

Programmtechnisch kann man es so lösen, dass bei einer Veränderung des Fensters die Breite des Listenfeldes dynamisch angepasst wird. Dazu bearbeitet man die **OnCanResize** Routine des Formulars (nicht der Listbox!):

```
procedure TForm1.FormCanResize(Sender: TObject; var NewWidth,
  NewHeight: Integer; var Resize: Boolean);
begin
  Listbox1.Width:=NewWidth-Listbox1.Left-32;
end;
```

Die neue Breite der Listbox ist die neue Breite des Formulars, abzüglich der linken Position der Listbox und 32 Pixeln für den Rand des Formulars (sieht gefälliger aus).

So könnte das Spiel aussehen:



Versuchen sie nun das Spiel zu verbessern:

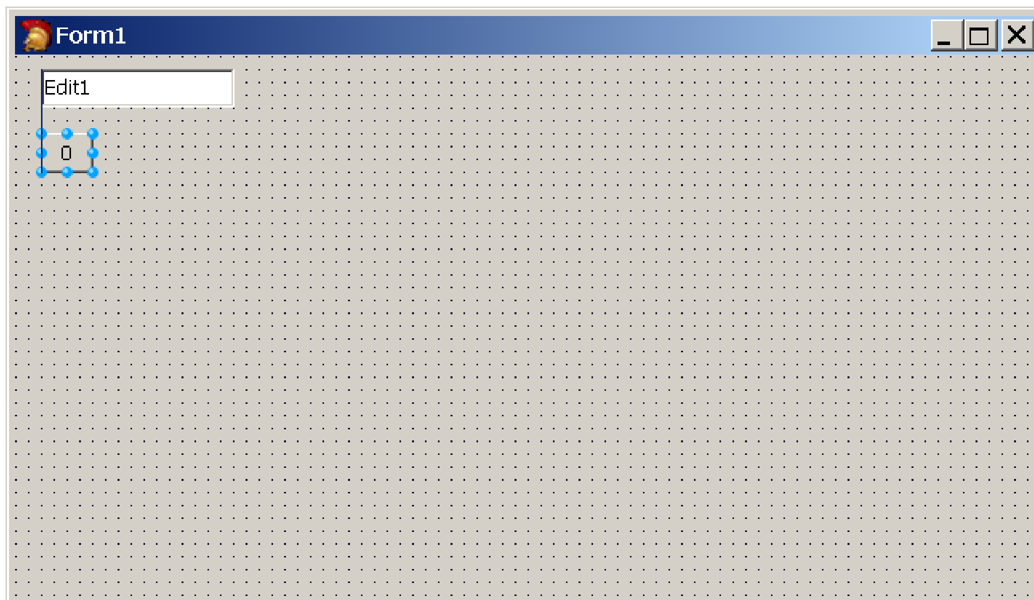
Geben sie in der Titelleiste (**Form1.caption**) jeweils aus, der wievielte Rateversuch es ist (Hinweis: Die Eigenschaft **items.count** der Listbox enthält die Anzahl der Elemente der Listbox)

Geben sie auch beim erfolgreichen Raten in die Listbox einen entsprechenden Eintrag aus.

Einführung in die Ereignisorientierte Programmierung.

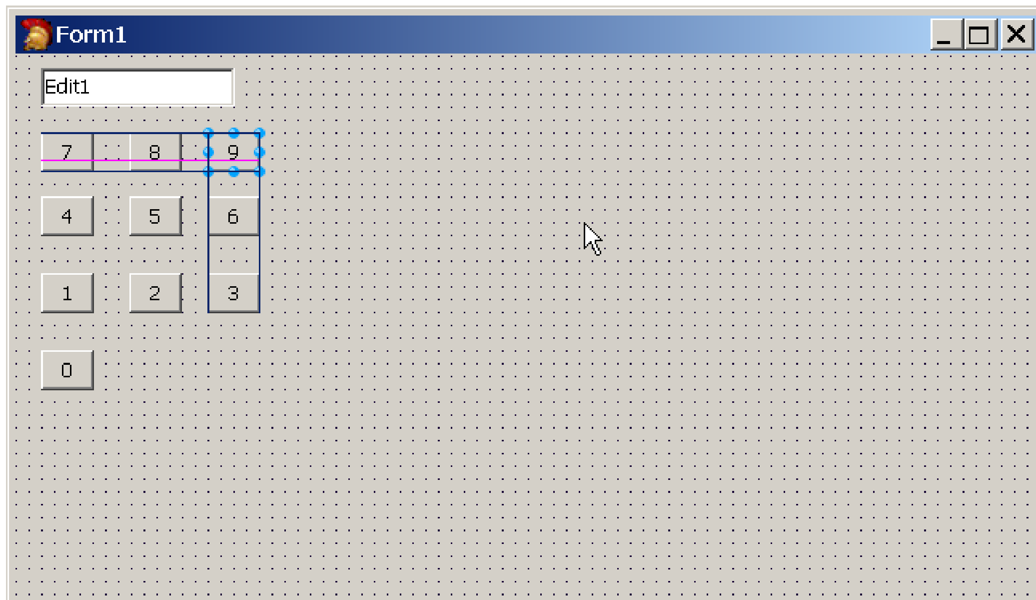
Heute wollen wir einen Taschenrechner programmieren. Dabei lernen wir die Programmierung in Delphi an einem Beispiel kennen.

Platzieren sie ein Editfeld (Tedit) oben und einen Button (Tbutton) darunter. Ändern sie Beschriftung (**Caption**) des Buttons im Objektinspektor in „0“ und verkleinern sie den Button, indem sie ihn an der rechten Seite anfassen und verkleinern. Ihr Formular sollte nun so aussehen:



Das Editfeld wird unser Rechenergebnis aufnehmen. Sie können, wenn sie keine Eingabe zulassen wollen, die Eigenschaft **Readonly** im Objektinspektor auf true setzen. Dann wird die Ausgabe gegraut und es ist keine Eingabe möglich.

Der Button dient zur Eingabe der Ziffer „0“. Wir benötigen mehrere dieser Buttons um die anderen Zahlen eingeben. Zu können. Markieren sie nun den Button und drücken sie STRG-C: Der Button ist in die Zwischenablage kopiert worden. Fügen sie ihn nun neunmal ein (STRG-V) und ändern sie jeweils die Beschriftung und Position, sodass sie Folgendes erhalten:



Wichtig: Achten sie beim Einfügen auf die Reihenfolge und die Namen der Buttons. Der mit der „0“ beschriftete Button muss „Button1“ heißen und der mit „9“ beschriftete muss Button10 heißen. (Beschriftung = Button Nummer-1).

Im ersten Testlauf wollen wir dem Editfeld eine Ziffer anfügen, wenn wir auf einen Button klicken. Die wichtigste Eigenschaft des Editfeldes ist die Eigenschaft **„Text“**. Sie enthält den sichtbaren Text des Editfeldes.

Markieren sie den Button1 der mit „0“ beschriftet ist, gehen sie in den Objektinspektor und schreiben sie in die **OnClick** Routine:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
edit1.text:=Edit1.text+'0';
end;
```

Wenn sie nun das Programm laufen lassen, wird bei jedem Klick auf diesen Button eine „0“ an den Text im Editfeld angehängt. Die anderen Buttons sind noch ohne Funktion. Anstatt nun für jeden der zehn Buttons eine eigene Onclick Routine zu schreiben, ändern wir die Routine zuerst wie folgt ab:

```
procedure TForm1.Button1Click(Sender: TObject);

begin
if sender=button1 then edit1.text:=Edit1.text+'0';
end;
```

Markieren sie nun mit der Maus nun die anderen 9 Buttons und wählen sie im **Objektinspektor** das Onclick Ereignis für Button1 aus. Sie haben nun allen Button dasselbe Onclick Ereignis zugewiesen.

In der Routine wird eine 0 aber nur beim Klicken on Button1 angehängt. Der Parameter **Sender** enthält Informationen, wer die Routine aufrief, also die Komponente, mit der die Routine verknüpft ist. Also Button1 bis Button10. Das erlaubt es nur eine Routine für mehrere Komponenten zu verwenden und dort jeweils durch Abfrage des Parameters „Sender“ festzustellen, wer die Routine aufrief. Durch die **If** Abfrage wird nun eine „0“ nur bei Button1 angehängt. Kopieren sie die Zeile mit dem If THEN und fügen sie diese neunmal ein und ändern sie diese dann wie folgt ab:

```
begin
```

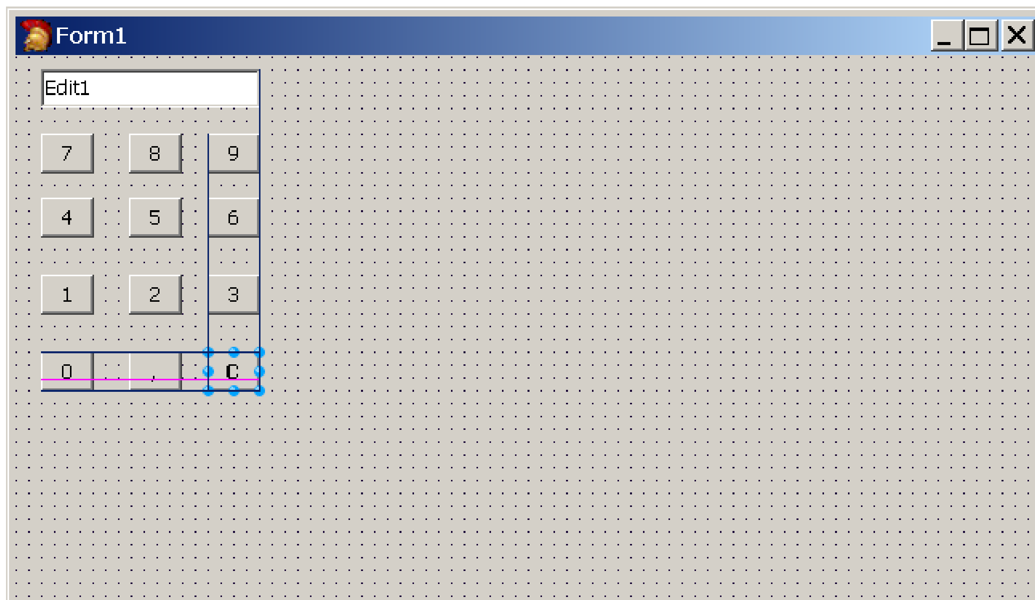
```
if sender=button1 then edit1.text:=Edit1.text+'0';
if sender=button2 then edit1.text:=Edit1.text+'1';
if sender=button3 then edit1.text:=Edit1.text+'2';
if sender=button4 then edit1.text:=Edit1.text+'3';
if sender=button5 then edit1.text:=Edit1.text+'4';
if sender=button6 then edit1.text:=Edit1.text+'5';
if sender=button7 then edit1.text:=Edit1.text+'6';
if sender=button8 then edit1.text:=Edit1.text+'7';
if sender=button9 then edit1.text:=Edit1.text+'8';
if sender=button10 then edit1.text:=Edit1.text+'9';
end;
```

Lassen sie das Programm laufen. Nun werden alle Zahlen angehängt. Achten sie darauf, ob die Button Beschriftungen auch mit den Namen korrespondieren, sonst könnte es anders laufen als geplant.

Übung:

Schreiben sie die Routine so um, dass sie eine CASE-Abfrage benutzen.

Nun fehlt noch ein Button für das Dezimalkomma, und einer um die Eingabe zu löschen. Fügen sie zwei Buttons ein, sodass die Oberfläche wie folgt aussieht:



Das Feld mit „C“ erhalten sie in Fetter Schrift, wenn sie in die **Font** Eigenschaft klicken und im Dialog dann „**Fett**“ markieren.

Für das Komma Feld wird auch dieselbe Onclick Routine wir für die Zahlen verwendet. Wir können diese im Objektinspektor auswählen, wenn wir im Onclick Ereignis unter den schon vorhandenen Routinen auswählen. Für das Einfügen des Dezimalkommas ergänzen wir die Button1Click Routine wie folgt:

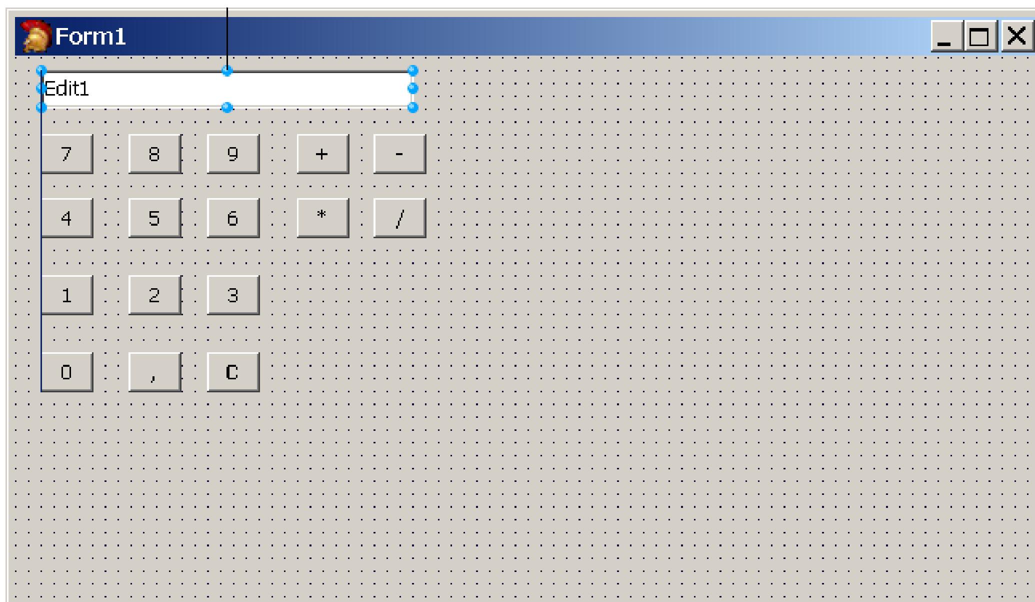
```
procedure TForm1.Button1Click(Sender: TObject);
begin
if sender=button1 then edit1.text:=Edit1.text+'0';
if sender=button2 then edit1.text:=Edit1.text+'1';
if sender=button3 then edit1.text:=Edit1.text+'2';
if sender=button4 then edit1.text:=Edit1.text+'3';
if sender=button5 then edit1.text:=Edit1.text+'4';
if sender=button6 then edit1.text:=Edit1.text+'5';
if sender=button7 then edit1.text:=Edit1.text+'6';
```

```
if sender=button8 then edit1.text:=Edit1.text+'7';
if sender=button9 then edit1.text:=Edit1.text+'8';
if sender=button10 then edit1.text:=Edit1.text+'9';
if sender=button11 then edit1.text:=Edit1.text+',';  // Neu
end;
```

Für die Löschroutine („C“ Taste) schreiben wir ein eigenes OnClick Ereignis:

```
procedure TForm1.Button12Click(Sender: TObject);
begin
edit1.text:='';
end;
```

Nun funktioniert das Eingeben von Zahlen. Nun wird es an der Zeit die Grundrechenarten zu definieren. Zuerst fügen wir die Buttons dafür ein:



Beim Klicken auf eine der Buttons für das Rechnen bei einem echten Taschenrechner passiert Folgendes:

- Die Anzeige wird gelöscht
- Der Taschenrechner merkt sich die eingegebene Ziffer
- Der Taschenrechner merkt sich die Operation
- Nach Eingabe einer zweiten Ziffer wird beim Drücken auf „=“ die Rechnung durchgeführt und das Ergebnis angezeigt.

Zum Merken der gedrückten Taste benutzen wir einen Aufzählungstyp. Ergänzen sie nach dem Schlüsselwort **Type** folgendes:

```
type
  Operation = (plus,minus,mal,geteilt);  // Neu
  TForm1 = class(TForm)
  ....
```

Der Typ Operation hat nur vier mögliche Werte, die in den Klammern angegeben wurden. Nun brauchen wir noch eine Variable, welche diesen Typ als Datentyp hat, um uns die aktuelle Operation zu merken. Diese fügen wir bei private ein:

```
  ...
  private
    merk: operation; // Neu
    { Private-Deklarationen }
  public
    ....
```

Weiterhin brauchen wir noch eine Variable, welche die erste Zahl aufnimmt. Auch diese führen wir bei **private** ein:

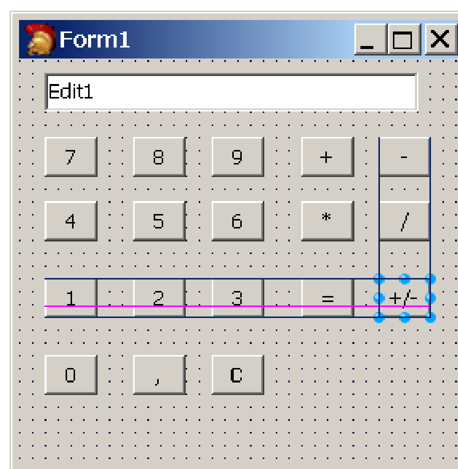
```
  ...
  private
    merk: operation;
    zahl: double;
    { Private-Deklarationen }
  public
    ....
```

Nun können wir für alle vier Rechenarten gemeinsam wieder in eine Ereignisbehandlungsroutine schreiben. Markieren sie die vier Buttons und schreiben sie dann folgende OnClick Routine.

```
procedure TForm1.Button13Click(Sender: TObject);

begin
  zahl:=strtofloat(edit1.text);
  if sender=button13 then merk:=plus;
  if sender=button14 then merk:=minus;
  if sender=button15 then merk:=mal;
  if sender=button16 then merk:=geteilt;
end;
```

Sie Routine **StrToFloat** konvertiert einen String in eine Fließkommazahl. Es gibt dasselbe auch für Ganzzahlen (Integer : **StrToInt**) und es geht auch in die umgekehrte Richtung (**IntToStr** und **FloatToStr**).

Nun fehlt noch das Gleichheitszeichen um die Rechnung durchzuführen und das Vorzeichen. Fügen sie zwei Buttons hinzu:



Zuerst die Behandlung für den Vorzeichen Button. Sie ist etwas kompliziert. Wir müssen zuerst einmal testen, ob der String in **Edit1.text** nicht leer ist. Erst dann können wir testen, ob das erste Zeichen ein Minus ist. (Sonst erhalten wir einen Fehler, weil man bei einem leeren String natürlich nicht auf das erste Zeichen zugreifen kann.)

Wenn das erste Zeichen ein Minus ist, dann weisen wir Edit1.Text eine Kopie des Editfeldes ab dem zweiten Zeichen zu (damit löschen wir das Minuszeichen), ansonsten fügen wir links ein Minus ein. Das führt uns zwanglos zu folgendem Code:

```
procedure TForm1.Button18Click(Sender: TObject);
begin
if (length(edit1.text)>0) then
if (Edit1.text[1]<>'-') then edit1.text:='-'+edit1.text else
edit1.text:=copy(edit1.text,2,20);
end;
```

Die Routine für das Gleichheitszeichen ist dagegen einfach: Wir haben uns ja die Operation gemerkt und können nun die Berechnung in einem Case Fall durchführen:

```
var zahl2 : double;

begin
  zahl2:=strtofloat(edit1.text);
  case merk of
    plus: zahl:=zahl+zahl2;
    minus: zahl:=zahl-zahl2;
    mal: zahl:=zahl*zahl2;
    geteilt: zahl:=zahl/zahl2;
  end;
  edit1.text:=floattostr(zahl);
end;
```

Wenn wir das Programm nun laufen lassen, so gibt es noch einige Schönheitsfehler. Der Erste ist der, dass beim Start noch der Text „Edit1“ im Editfeld steht. Das kann man lösen, indem man die Text Eigenschaft im Objektinspektor ändert.

Der Zweite ist schwerwiegender: Nach jeder Berechnung werden neue Ziffern an das alte Ergebnis angehängt, sodass wir erst das Eingabefeld nach einer Berechnung löschen müssen.

Dies lösen wir folgendermaßen: Wir merken uns, wann neu eingetippte Ziffern das Eingabefeld löschen sollten. Das ist der Fall, wenn

- Eine Operationstaste getippt wurde (+ - / *)
- Auf „=“ getippt wurde.

Wir brauchen dazu eine boolesche Variable, die wir im private-Teil deklarieren:

```
  private
    merk: operation;
    zahl: double;
    neue_zahl : boolean;
    { Private-Deklarationen }
  public
```

Diese muss, wenn wir auf = oder + - / * tippen, auf true gesetzt werden:

```
procedure TForm1.Button13Click(Sender: TObject);

begin
  zahl:=strtofloat(edit1.text);
  if sender=button13 then merk:=plus;
  if sender=button14 then merk:=minus;
  if sender=button15 then merk:=mal;
  if sender=button16 then merk:=geteilt;
  neue_zahl:=true;  // Neu!!!
end;

procedure TForm1.Button17Click(Sender: TObject);
```

```
var zahl2 : double;

begin
  zahl2:=strtofloat(edit1.text);
  case merk of
    plus: zahl:=zahl+zahl2;
    minus: zahl:=zahl-zahl2;
    mal: zahl:=zahl*zahl2;
    geteilt: zahl:=zahl/zahl2;
  end;
  edit1.text:=floattostr(zahl);
  neue_zahl:=true;  // Neu!!!
end;
```

und wenn wir nun eine Ziffertaste drücken, sollte die Eingabe gelöscht werden, wenn die Variable **neue_zahl=**true ist und danach muss sie natürlich auf false gesetzt werden – sonst wird die Eingabe bei jeder Ziffer gelöscht:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if neue_zahl then edit1.text:=''; // Neu
  neue_zahl:=false;                  // auch neu!
  if sender=button1 then edit1.text:=Edit1.text+'0';
  if sender=button2 then edit1.text:=Edit1.text+'1';
  if sender=button3 then edit1.text:=Edit1.text+'2';
  if sender=button4 then edit1.text:=Edit1.text+'3';
  if sender=button5 then edit1.text:=Edit1.text+'4';
  if sender=button6 then edit1.text:=Edit1.text+'5';
  if sender=button7 then edit1.text:=Edit1.text+'6';
  if sender=button8 then edit1.text:=Edit1.text+'7';
  if sender=button9 then edit1.text:=Edit1.text+'8';
  if sender=button10 then edit1.text:=Edit1.text+'9';
  if sender=button11 then edit1.text:=Edit1.text+',';
end;
```

Der Taschenrechner ist nun fertig und rechnet. Er ist noch nicht perfekt. So können sie das Komma mehrmals eingeben. Dadurch gibt es einen Fehler beim Konvertieren mit **StrToFloat**. Ändern sie dies manuell ab. (Hinweis: Die Funktion **Pos(Suchstring,Zielstring)** liefert die Position von Suchstring in Zielstring zurück oder 0 wenn der Suchstring nicht im Zielstring vorhanden ist).

Aufgaben

Versuchen sie folgendes selbstständig zu lösen. Wir arbeiten dann an dem Taschenrechner weiter.

- Vervollständigten sie den Taschenrechner so, dass verschiedene jetzt noch mögliche Fehler nicht auftreten wie: doppeltes Dezimalkomma, Weiterrechnen bei mehrmaligem Druck auf die „=“ Taste.
- Implementieren sie eine Prozentrechnung mit der „%“ Taste. Einige Beispiele:

  100 + 10 % = 110

  100 – 40 % = 60

  100 * 5 % = 5
- Wie kann man den Taschenrechner um Funktionen (Sin, Cos, Ln, Exp) erweitern? Programmieren sie eine der Funktionen (Sin oder Cos z.B.)
- Wie vermeiden sie einen Fehler bei der Division durch 0? Geben sie dann im Textfeld „Error“ aus.

Theorielteil

Nachdem sie nun schon etwas Erfahrung mit Komponenten haben, sollten sie im Praxisteil selbst die Eigenschaften einfacher Komponenten ändern können. Daher an dieser Stelle nur eine Beschreibung der verwendeten Komponenten.

Die **Checkbox** ist eine sehr einfach zu verwendende Komponente. Wichtig ist neben den allgemeinen Eigenschaften wie **Caption, Enabled** und **Font** eigentlich nur eine:

Die Eigenschaft **Checked** ist eine Boolean Variable. Ist die Checkbox markiert, so hat sie den Wert true, sonst den Wert false.

Die Radiobuttons gibt es als einzelnen Button, der aber keinen praktischen Nutzen hat. Sinnvoll sind sie nur als Gruppe. Die **Radiogroup** Komponente hat zwei wichtige Eigenschaften: Das eine ist die Variable **ItemIndex**. Sie enthält den Radiobutton, der gerade markiert ist. Sie ist 0 beim Ersten, 1, beim Zweiten etc. Ist keiner markiert, so beträgt der Wert von ItemIndex -1.

Die Anordnung kann man über die Zahl der Spalten (**Column**) festlegen. Standardmäßig wird in einer Spalte angeordnet.

Komplexer ist die Eigenschaft Items. Items ist eine Stringliste, ein dynamisches Array von Strings. Jeder Eintrag steht für eine Zeile. Die Stringliste selbst hat wiederum einige Subeigenschaften, auf die sie zugreifen können, wenn sie einen Punkt an Items anhängen.

Die Wichtigsten sind:

- **Count**: Anzahl der Elemente
- **[nn]**: Zugriff auf ein einzelnes Element. Der Index nn läuft von 0 bis Count-1. Wichtig! Nicht von 1 bis Count!
- **Add (String)**: Hinzufügen eines weiteren Eintrags
- **Delete (Integer)**: Löschen eines Elementes
- **IndexOf (String): Integer:** Liefert den Index eines Strings in der Liste dazu.
- **Text**: alle Strings als ein Text
- **Savetofile(String)**: Speichert die Strings in einer Datei.
- **Loadfromfile(String)**: Lädt die Strings von einer Datei.
- **Clear**: Löscht die Liste.

Beispiele:

```
Radiogroup1.items.add ('Exponentialdarstellung');
Listbox1.items.clear;
Listbox1.Items [4]:= '47.11';
Listbox1.items.add (Edit2.Text);
```

Stringlisten sind die am häufigsten benutzten Objekte in Delphi. Sie werden auch beim nächsten Element, der Listbox benutzt.

Die **Listbox** hat auch die Eigenschaft **Items**. Auch sie enthält eine Stringliste. Es sind die sichtbaren Elemente. **Itemindex** enthält das jeweils selektierte Element. Es ist aber auch möglich, mehrere Elemente zu selektieren. Dazu gibt es die Eigenschaften **MultiSelect** (erlaubt das Selektieren von Elementen direkt nebeneinander mit Shift) und **ExtendedSelect** (erlaubt das Selektieren von Elementen, die nicht nebeneinander sind, mit STRG). Dies kann aktiviert werden, indem diesen Boolean Variablen jeweils True zugewiesen wird.

Sind mehrere Einträge aktiviert, so muss man auf die Eigenschaft **Selected** der Listbox zurückgreifen:

Selected[nn] hat einen Boolean Wert: False wenn nicht selektiert, true wenn selektiert. Der Index nn läuft wie bei Items von 0 bis items.count-1.

Die Format Routine

Es gibt auch eine Routine, welche es erlaubt komfortabler als es mit Writeln möglich ist, Ergebnisse zu formatieren. Die **Format** Funktion. Die Syntax ist:

Format (Formatstring, [Liste von Ausdrücken]): String

Der Formatstring ist ein String, bei dem bestimmte Formatierungsinformationen enthalten sind. Alle anderen Zeichen im String werden in das Ergebnis kopiert.

Es gibt eine Reihe von Optionen, die in der Hilfe näher erläutert sind. Alle beginnen mit dem Prozent-Zeichen **%** und die Option hat einen Buchstaben. Für unser Projekt benötigen wir folgende:

%e: Exponentialdarstellung

%g: Allgemeine Darstellung (wähle die aus, die am kürzesten ist)

%.2f: Darstellung mit Festkomma und 2 Nachkommastellen

%x: hexadezimale Darstellung: Funktioniert nur mit Integerzahlen. Sie müssen eine Fließkommazahl mit Trunc () in eine Integerzahl umwandeln, bevor sie diese ausgeben.

Die Ausdruckliste (in eckigen Klammern) sind Variablen und mathematische Ausdrücke. Sie müssen zu dem String passen. Wenn im String z.B. zwei Zahlen formatiert werden sollen, so muss es zwei Ausdrücke geben. Fehler bei falschen Parametern oder zu wenigen Parametern treten bei der Format Funktion erst während der Programmausführung auf.

Bsp:

gegeben: Zahl als Double Variable 3.3376464

```
Format ('%f.2',[zahl]) ergibt '3.34'
Format ('Ergebnis: %g ',[zahl]) ergibt '3.3376464'
Format ('%e %x',[zahl, Trunc (zahl)]) ergibt '3.33e+01  3'
```

Aktionen zu Programmbeginn und Ende

Bislang kannten sie Ereignisse, die vom Anwender ausgelöst wurden. Es gibt aber auch Ereignisse, die von Windows ausgelöst werden. Für uns heute wichtig sind die **OnCreate** und **OnDestroy** Routinen. Beide sind nur für das Formular definiert und werden ausgelöst, wenn das Formular erzeugt und zerstört wird (z.b., wenn wir es schließen).

Diese Routinen sind ideal um Daten zu laden und zu speichern, z.B. Eingaben, die beim nächsten Programmstart wieder vorhanden sein sollen.

Die Zwischenablage

Es ist möglich, Text in die Zwischenablage zu kopieren. Dazu muss die **uses** Liste am Anfang des Codes um die Unit **clipbrd** ergänzt werden. Danach steht die Zwischenablage den Programmen zur Verfügung.

Das Kopieren eines Textes z.B. des Strings „mytext“ in die Zwischenablage geht mittels folgender Zeile:

```
clipboard.SetTextBuf(pchar(mytext));
```

Einen Text aus der Zwischenablage können sie so einem String zuweisen:

```
var x : String;

begin
  x:=Clipboard.AsText;
end;
```

Viele Komponenten haben Routinen wie CopyToClipboard, CutToClipboard und PasteFromClipboard die etwas komfortabler sind. Dies ist z.b. bei den Editfeldern der Fall:

```
procedure TForm3.Edit1Click(Sender: TObject);
begin
  edit1.CopyToClipboard;
end;
```

Praxisteil

Heute erweitern wir den Taschenrechner, der in der letzten Stunde angefangen wurde. Wir wollen folgende Funktionalität einfügen:

- Wählbare Formate für die Darstellung des Ergebnisses (Fließkommaformat, festes Dezimalkomma, Hexadezimale Darstellung).
- Kopieren des Rechenergebnis in die Zwischenablage
- Merken der letzten Rechenergebnisse und Speichern dieser zum Programmende.

Aufgabe 1

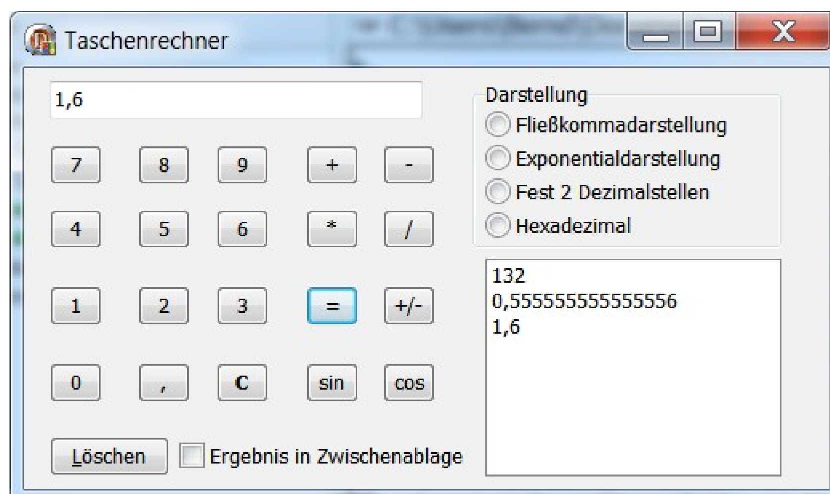
Ergänzen sie den Rechner aus der letzten Woche um eine Radiobox

Sie soll folgende Auswahlmöglichkeiten haben:

- Darstellung als Exponentialzahl
- Darstellung als in allgemeiner Form
- Darstellung als Festkommazahl mit 2 Dezimalstellen
- Darstellung als Hexadezimalzahl

Ändern sie die Ausgabe des Ergebnisses in Edit1.text so ab, dass die Auswahl berücksichtigt wird und der Text entsprechend formatiert wird. Hinweis: Sie benötigen dazu die **Format** Funktion und eine **Radiogroup.**

Ihr Taschenrechner könnte nun so aussehen:



Aufgabe 2

Wenn der Benutzer auf „=“ tippt und das Ergebnis anzeigt wird, soll es in einer Liste angefügt werden. Der Benutzer soll auch einen Eintrag in der Liste auswählen können, der dann im Editfeld erscheint und die derzeit eingetippte Zahl ersetzt. Natürlich muss dieser Eintrag auch bei der Rechnung mit berücksichtigt werden.

Hinweis: Sie brauchen dafür eine **Listbox**.

Aufgabe 3

Die Ergebnisse in der Liste müssen beim Beenden gespeichert und beim nächsten Programmstart geladen werden. Weiterhin brauchen sie eine Möglichkeit, sie zu löschen entweder alle auf einmal, oder wenn sie sich das zutrauen, die ältesten Einträge der Liste (die jüngsten 5 sollen erhalten bleiben).

Sie brauchen dazu die **LoadFromFile** und **SaveToFile** Routinen der Items Eigenschaft.

Achten sie darauf, dass sie in das Verzeichnis schrieben können, in dem Sie die Dateien ablegen. Da sie nur eine Datei einladen können die existiert, könnte es sein, dass sie die FileExists Funktion brauchen: **FileExists(Dateiname)** liefert true zurück, wenn die Datei existiert, ansonsten false.

Aufgabe 4

Damit auch andere Programme etwas von ihren Berechnungen haben, soll das Ergebnis auch in die Zwischenablage kopiert werden. Ob dies geschieht, soll der Anwender wählen können. Sie brauchen dazu die **Checkbox** und die Unit **clipbrd**.

Aufgabe 5 (für begabte oder fleißige Studenten)

Derzeit sind keine Kettenberechnungen möglich, wie z.B. 1+ 7 / 7 = ergibt 1 und nicht 1.1428... Da nach dem „=“ ausgewertet wird. Schreiben sie das Programm so um, dass Kettenberechnungen möglich sind.

Aufbau eines grafischen Programmes.

Wie schon erläutert ist ein grafisches Programm deutlich komplexer als die Konsolenanwendungen. Heute geht es darum, den Aufbau eines Formulars zu verstehen. Dabei gilt es mehrere Dinge zu verstehen, die gemischt in einem Delphi Programm vorkommen.

- Wo darf ich selbst Code einfügen, und wo sind Sektionen, die von der Entwicklungsumgebung verwaltet werden und an denen ich besser nicht herumfummele?
- Wie sind größere Programme im Allgemeinen aufgebaut?
- Was ist ein Objekt und wie werden grafische Oberflächen in Code umgesetzt?

Fangen wir mit dem Aufbau größerer Programme an.

Aufbau von Units

Bislang haben wir alles in eine Datei geschrieben. Das ist auch kein Problem, solange man nur ein kleines Programm entwickelt. Bei größeren Programmen empfiehlt es sich, die Funktionalität jedoch aufzugliedern. Bei grafischen Anwendungen ist dies nicht anders möglich, da alleine der Code für die grafischen Steuerelemente sehr umfangreich ist. Ein Programm besteht daher meistens aus einzelnen Teilen.

Pascal bietet dafür das Konzept der Units. Eine Unit ist eine Pascaldatei mit der Endung „.pas“ und einem definierten Aufbau:

```
Unit unitname;

interface

uses <Liste von anderen Units>

<öffentliche Deklarationen>

implementation

uses <Liste von anderen Units>

<lokale Deklarationen>
<Code>

initialization
// Code der beim Start ausgeführt wird (optional)
finalization
// Code der beim Ende ausgeführt wird (optional)
end.
```

Der Zweck von Units ist es den Programmcode in einzelne Teile mit einer definierten Funktion aufzuteilen. Sie kennen schon die Units „Sysutils“. Wie der Name andeutet, enthält sie Routinen des Systems, die nützlich sind, z.B. zur Umwandlung von Klein in Großbuchstaben. Der gesamte Code für die Verwaltung von Fenstern ist dagegen in der Unit „Forms“ enthalten.

Eine Unit zerfällt in zwei Teile. Der Teil zwischen **interface** und **implementation** ist der **öffentliche** Teil. Hier finden sie nur Deklarationen, aber keinen Code. Jede andere Unit welche diese benutzt kann auf Variablen, Konstanten und Typen in dieser Sektion zurückgreifen. Weiterhin finden sich dort die Köpfe von Prozeduren, welche im Implementations-Teil kodiert sind. Die Deklarationen im Interface Teil sind global im Programm. (Das Programm ist die Summe aller Units).

Sie haben bisher immer eine Unit benutzt: Sysutils die mit

```
uses
  SysUtils;
```

automatisch in ihren Quelltext eingebunden wurde. Damit konnten sie alle Routinen, die dort definiert wurden, benutzen. Das Gleiche gilt für Variablen und Konstanten wie z.B. Pi.

Der Deklarationsteil

Der öffentliche Teil der Unit wird auch Deklarationsteil genannt. Dies beschreibt seine Funktion: Er definiert Variablen, Typen und Konstanten, aber auch Prozeduren und Funktionen. Er beginnt mit dem Schlüsselwort **interface** und endet vor dem Schlüsselwort **implementation**.

Er enthält keinen ausführbaren Code. Daher gibt es einen Unterschied zwischen der Deklaration von Variablen, Typen und Konstanten auf der einen Seite und Prozeduren und Funktionen auf der anderen Seite.

Während Typen, Variablen und Konstanten nur einmal deklariert werden und dann in der ganzen Unit gültig sind, müssen Prozeduren und Funktionen **zweimal** deklariert werden.

Im Deklarationsteil schreibt man nur den Kopf, also die Kopfzeile von procedure / function mit dem Namen der Funktion(en) und Parametern. Damit ist bekannt wie die Funktion heißt und welche Daten sie benötigt. Den eigentlichen Code der Funktion benötigt man nicht um sie zu benutzen, aber die Definition muss bekannt sein. Daher steht in dem Deklarationsteil nur der Code.

Im Implementationsteil schreibt man dann den Code. Hier ein Beispiel:

```
unit demo;

interface

function Kreisumfang(radius : double) : double;

implementation

function Kreisumfang(radius : double) : double;

begin
  Result:=2*pi*radius;
end;

end.
```

Zwischen interface und Implementation steht der Prozedurkopf und nach implementation der Code mit dem Kopf. **Wichtig: Die Deklaration muss identisch sein.** Während sonst bei Prozeduren und Funktionen es egal ist, welchen Namen Parameter haben, muss hier der Name von Parametern identisch zur Deklaration sein. Am besten kopieren sie sich also die Zeile. Würden sie im Implementierungsteil schreiben:

```
function Kreisumfang(radius2 : double) : double;

begin
  Result:=2*pi*radius;
end;
```

würde einen Fehler geben, weil in der Deklaration der Parameter „radius“ und in der Implementierung „radius2“ heißt.

Die Funktion des Deklarationsteiles ist es, die Teile der Unit anzugeben, die andere Programme benutzen können. Dafür müssen diese aber nicht die Implementierung, also den eigentlichen Code kennen. Es reicht, wenn sie wissen, wie man sie aufruft und welche Parameter sie entgegen nehmen. Das erlaubt es auch die Units in übersetzter Form (mit der Dateiendung .dcu) weiterzugeben: Sie enthalten die komplette Information

des Deklarationsteils, aber der Implementationsteil ist in Maschinencode übersetzt worden und liegt nicht als Pascal Quelltext vor.

Der Implementationsteil

Im Implementationsteil (der dem Schlüsselwort implementation folgt) wird zum einen der Code der Funktionen ausgeführt, welche im Deklarationsteil definiert wurden. Er kann aber auch eigene Variablen, Konstanten, Typen und Funktionen/Prozeduren enthalten. Im Implementationsteil gemachte Deklarationen und Prozeduren/Funktionen sind nur innerhalb der Unit bekannt. Sie sind global in dieser Datei. (Das entspricht dem Verhalten, dass sie von Konsolenprogrammen kennen).

Der Implementationsteil endet mit einem begin … end. Wie das Hauptprogramm eines Konsolenprogrammes. Anders als bei diesem kann aber das begin weggelassen werden.

Wenn nun bestimmte Dinge beim Start oder Ende durchgeführt werden müssen (Dateien geöffnet, Arrays initialisiert oder Verzeichnisse ermittelt werden), so können diese in zwei Blöcken geschrieben werden, die sich im Hauptprogramm (am Ende) einer Unit befinden.

Der erste Block beginnt mit dem Wort **initialization** und endet entweder mit dem Wort **finalization** oder dem **end.** des Hauptprogramms. Dieser Block wird ausgeführt, wenn das Programm gestartet wird, bevor die erste Anweisung des Hauptprogrammes ausgeführt wird. Gibt es mehrere Units, so werden die **initialization** Blöcke nacheinander ausgeführt.

Das Zweite ist der **finalization** Teil. Er endet mit dem end. Des Hauptprogramms. Er wird durchlaufen, wenn das Hauptprogramm beendet wird. Alle Anweisungen in diesem Block werden ausgeführt, wenn das Hauptprogramm beendet wird. Der finalization Teil kommt immer nach dem initialization Teil.

Nutzen sie diese beiden Blöcke um Initialisierungen durchzuführen, wie z.B. Variablen auf Startwerte zu setzen oder Aufräumarbeiten zu erledigen.

Damit ist eine Unit definiert. Wie benutzen sie nun eine Unit mit **Uses**, machen sich also die Funktionalität zu eigen? Dazu fügt man nach dem Schlüsselwort „uses“ alle Units (getrennt durch Kommas) auf, deren Routinen man benötigt. Uses wird dabei direkt nach den Schlüsselworten „implementation“ oder „interface“ geschrieben, vor jeder anderen Deklaration oder Code. Beim Hauptprogramm (Endung .dpr) kommt die Liste von Uses ebenfalls vor Deklarationen und Code.

Bei Units haben sie so zwei Möglichkeiten mit uses andere Units einzubinden – einmal im implementation Teil und einmal im Interface Teil. Warum nun zwei uses Klauseln? Die Uses Klausel im interface Teil dient dazu Units bekannt zu machen, die benötigt werden für die Deklarationen im Interface Teil der aufzurufenden Unit. Sie könnten z.B. Farben definieren und dazu den Typ TColor benutzen, um dies mit Worten zu tun (clblue,clgreen,clyellow...) Dazu brauchen sie die Definition des Typs TColor, die sie in der Unit „Graphics“ finden. Damit ihre Deklaration im Interface Teil korrekt übersetzt werden kann, müssen sie vorher diese Unit einbinden.

Das ist immer der Fall für die verschiedenen grafischen Elemente eines Formulars. Daher enthält die Uses Liste eines Formulars sehr viele Units, die für die Definition des Formulars benötigt werden.

Es kann aber auch sein, dass sie eine Unit nur beim Schreiben des Codes brauchen. Sie könnten z.B. die Funktion UpperCase() aus Sysutils benötigen. Dann nutzen sie die Uses Liste im implementationsteil. Das hat auch den Vorteil, das so zirkuläre Referenzen (Unit A benötigt Unit B zur Übersetzung und Unit B wiederum Unit A) vermeiden werden können.

Aufbau eines grafischen Programmes

Mit dem was sie nun über Units wissen können sie schon einen Teil eines typischen grafischen Programms verstehen.

```
unit Unit1;        // Unitname

interface          // Deklarationsteil

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;        // benötigte Units um das Formular zu erstellen

type
  TForm1 = class(TForm)
  private
    { Private-Deklarationen }
  public
    { Public-Deklarationen }
  end;

var
  Form1: TForm1;

implementation      // Implementierung des Formulars

{$R *.dfm}

end.                // Ende des Hauptprogramms
```

Nun folgt noch die Erklärung des Formulars an sich. Eine grafische Oberfläche hat zwei wesentliche Unterschiede zu ihren Konsolenprogrammen:

Ihr Konsolenprogramm wird so durchlaufen, wie sie geschrieben haben. Bei einem grafischen Programm steuern sie nicht das Programm durch den geschriebenen Code, sondern reagieren auf Ereignisse, die der Anwender auslöst.

Eine grafische Oberfläche ist ein Objekt.

Was ist ein Objekt? Im wahren Leben haben sie dauernd mit Objekten zu tun. Ein Objekt hat Eigenschaften (dies sind auf den Computer übertragen die Daten) und es kann Dinge tun (das entspricht dem Code). So wurde dieses Konzept auf den Computer übertragen. Objekte sind Daten, verbunden mit dem Code um etwas mit den Daten anzufangen. Dadurch ergeben sich weitergehende Möglichkeiten. So können sie bei einem Array z.b. vor jedem Zugriff prüfen, ob der Index korrekt ist.

Eine grafische Delphi Anwendung besteht aus zwei Teilen. Im Deklarationsteil der Unit (nach interface) finden sie die Definition eines Fensters mit allen Bestandteilen, die sie hinzugefügt haben.

Diese beginnt zuerst mit

```
type
  TForm1 = class(TForm)
```

Damit ist klar, das Tform1 ein Datentyp ist. Alles was nun noch bis zum **end;** folgt ist Bestandteil der Definition dieses Datentyps.

Sie enthält weiterhin drei Sektionen, getrennt durch die Schlüsselworte public und private.

Vor **public** ist die Sektion die das System verwaltet. Wenn sie eine Komponente hinzufügen (z.B. ein Editfeld) oder eine Ereignisbehandlungsroutine schreiben (z.B. ein OnClick Ereignis), dann finden sie dort den neuen Kopf der Routine oder die Deklaration der Komponente eingefügt. Die Reihenfolge ist in allen Sektionen gleich: erst alle Komponenten (oder Variablen) und dann alle Routinen.

Das Schlüsselwort public leitet zu den zwei Sektionen über in denen Sie selbst eigene Variablen und Routinen einbinden können. Die zweite Sektion beginnt mit private. Auch hier ist die Reihenfolge: Zuerst eigene Variablen, dann eigene Routinen.

Was ist der Unterschied zwischen public und prviate? Auch hier hat man sich am täglichen Leben orientiert. „private“ dient dazu Eigenschaften und Routinen einzubinden, die intern (in diesem Formular) benötigt werden. Public wird für Routinen verwendet die auch andere Formulare (sie können eine Anwendung aus mehreren Fenstern konstruieren) benutzen können.

Dazu ein Vergleich aus dem täglichen Leben. Ein Auto hat Eigenschaften (es hat eine Farbe, Anzahl der Sitze, PS) und Fähigkeiten (beschleunigen, bremsen). Die meisten davon sind für den Käufer gedacht, aber einige auch nur für die Werkstatt (Diagnose durchführen, Fehlerspeicher auslesen etc.). Dies ist in etwa das gleiche Konzept, wie hinter private und public steht.

Mit dem Schlüsselwort **End;** wird die Definition des Typs eines Fensters beendet. Damit existiert dieses jedoch nur als Typ. Die Variable wird weiter unten definiert.

Var Form1: Tform1;

Dies ist eine klassische Variablendefinition: Es wird die Variable mit dem Namen Form1 vom Datentyp Tform1 definiert, also gerade dem Typ, der weiter oben angegeben ist.

Alle Zugriffe auf Methoden und Eigenschaften eines Objektes gehen dann in der Form **Variablenname.Methode** also z.B.

Form1.Edit1.Text:='Hallo';

Ein häufiger Anfängerfehler ist es, dafür den Typ zu nehmen und zu schreiben:

TForm1.Edit1.Text:='Hallo';

Das ergibt einen Fehler. Innerhalb der Routinen des Typs (also den von ihnen erstellten Ereignisbehandlungsroutine, wie OnClick) ist es möglich den Variablennamen wegzulassen, da diese sich dann immer auf das aktuelle Formular beziehen:

Edit1.Text:='Hallo';

wäre z.B. in einer OnClick Routine gültig.

Im Implementierungsteil finden sie dann die Routinen, die in der Deklaration definiert wurden. Das sind zum einen die Routinen, die das System für sie anlegt (wenn sie z.B. eine OnClick Routine implementieren. Wenn sie eine eigene Routine schreiben, so müssen sie diese sowohl in der Deklaration wie auch in der Implementierung einfügen.

Jede Routine hat eine bestimmte Form der Benennung und zwar:

procedure Typenname.Routinenname(<parameter>);

bzw.

function Typenname.Routinenname(<parameter>) : <Rückgabewert>;

Der Typenname ist in der Deklaration definiert. Im obigen Fall ist es Tform1. Dann folgt nach einem Punkt der eigentliche Name der Prozedur. Die Funktion/Prozedur ist teil dieses Typs. Auch hier gilt: Innerhalb von Prozeduren dieses Typs können sie dies weglassen.

Übungsaufgaben

Folgende Übungsaufgabe muss während des Semesters selbstständig als Vorbereitung für den benoteten Programmentwurf gelöst und zwei Wochen vor der letzten Vorlesungsstunde per Mail zugesandt werden (benötigte Dateien: alle mit der Endung *.pas, *.dfm und *.dpr in eine Zip-Datei packen)

Die Gruppeneinteilung orientiert sich an dem letzten Semester. Wenn sie in der Praxisphase in der Gruppe A waren, so lösen sie nun die Aufgabe für die Gruppe A. Analoges gilt für die anderen Gruppen.

Mailadresse: bl@bernd-leitenberger.de

Gruppe A

Tic-Tac-Toe ist ein Spiel, bei dem zwei Spieler abwechselnd auf einem 3x3 Felder großen Spielfeld ein Kreuz oder einen Kreis setzen. Sieger ist, wer zuerst drei eigene Symbole in einer Reihe (längs, quer,diagonal) hat. Die Regeln von Tic Tac Toe finden sie z.B. in der Wikipedia:

http://de.wikipedia.org/wiki/Tic_Tac_Toe

Ihre Aufgabe: Programmieren sie eine grafische Anwendung bei der sie tic Tac Toe gegen den Computer spielen können. Der Computergegner soll soweit intelligent sein, dass er auch gewinnen kann, wenn sie einen Fehler machen.

Gruppe B

Sie haben die Aufgabe für ihren Betrieb, der Panzerhaubitzen für die Bundeswehr fertigt ein Programm zu schreiben, mit dem die Soldaten recht einfach die Entfernung berechnen können, in der eine Granate aufschlägt. Gegeben ist die Geschwindigkeit Vo mit der die Granate das Rohr verlässt (in m/s), die Neigung des Rohrs zu horizontalen (α), der Rückenwind Vw (in km/h) und der Cw Wert der Granate.

Die Zusammenhänge sind nun folgende:

Die Geschwindigkeit direkt nach Verlassen des Rohrs splittet sich auf in eine Geschwindigkeit in der Horizontalen (Weite) und der Vertikalen (Höhe):

Vh0= cos (α)*V0

Vv0 = sin(α)*V0

Vh, die momentane Horizontalgeschwindigkeit, errechnet sich durch Addieren der Horizontalgeschwindigkeit vh0 nach des Verlassen des Rohrs und des Rückenwindes über die Zeit.

Vh = Vh0 + Vw

Vv, die momentane Vertikalgeschwindigkeit, nimmt durch die Gravitation ab:

Vv = Vv0 – g

g, die Erdbeschleunigung ist wiederum von der Höhe abhängig:

$g = 5{,}976E24 * 6{,}6726E\text{-}11 / (h+6371000)^2$

mit:

5,976E24: Erdmasse in kg

6,6726E-11: universelle Gravitationskonstante

6371000: mittlerer Erdradius in m.

Die Granate erreicht eine Gipfelhöhe, wenn Vv=0, danach nimmt die Höhe ab, da Vv durch die Gravitation negativ wird. Sie schlägt auf dem Erdboden auf, wenn h<=0.

Von Bedeutung für beide Geschwindigkeiten ist noch der Luftwiderstand, er reduziert die Geschwindigkeit nach:

$v = v - v*cw*d$

Der cw Wert liegt bei Granaten typischerweise bei 0,03 bis 0,08. Die Berechnung ist sowohl mit vh wie mit vv zu machen.

Die Dichte der Atmosphäre ist am Boden 1 und nimmt mit der Höhe ab:

$d = 1 * \exp(-h/8400)$

(8400 m: Skalenhöhe für Halbierung des Atmosphärendrucks in der Modellatmosphäre)

Der Weg in beiden Richtungen (Weite, bzw. Höhe) erhält man durch Summation der Geschwindigkeiten über die Zeit:

$w = \int Vh$

$h = \int Vv$

Schreiben sie ein Programm, das hinreichend genau die Weite eines Schusses durch Simulation der Flugbahn über die Zeit berechnet. Geben sie auch die Aufschlagsgeschwindigkeit aus. Diese ist errechenbar nach: $v = \sqrt{(vv^2+vh^2)}$.

Benötigte Funktionen: Sin, Cos, Exp ($=e^x$), Sqrt ($=\sqrt{x}$) und Sqr (x^2).

Bei Sinus und Cosinus ist darauf zu achten, dass diese im Bogenmaß definiert sind.

Optional: Im Normalfall sind die Soldaten daran interessiert ein Ziel zu treffen, dessen Entfernung sie kennen. Nutzen sie die Leistungsfähigkeit des Computers, um iterativ den Abschusswinkel zu berechnen (die einzige Größe, die vom Soldaten beeinflusst werden kann), wenn V0,Vw und cw der Granate bekannt sind. Die Genauigkeit heutiger Geschütze beträgt 20 m. Die Simulation kann also abgebrochen werden, wenn die berechnete Weite innerhalb von 20 m an der gewünschten Weite liegt.

Gruppe C

In der Fertigung ihres Betriebes gibt es zahlreiche Werkzeuge mit Verschleißteilen. Ab einer bestimmten Dauer werden Schnittkanten ungenau, erhalten ausgestanzte Teile mehr Spiel oder sitzen Muttern nicht mehr sauber auf den mit einer anderen Maschine hergestellten Schraube.

Die Vorgehensweise, um zu ermitteln, wenn Verschleißteile ausgewechselt werden müssen, ist es Proben auszumessen und dann die Standardabweichung mit der maximal zulässigen Abweichung zu vergleichen. Ein Teil muss ausgewechselt werden, wenn die Standardabweichung die 50% der Toleranz überschreitet.

Definition

Mittelwert xm = $$x_m = \frac{\sum_{i=1}^{n} x_i}{n}$$

Beispiel: Der Mittelwert aus 4.4, 5.0, 5.6 ist 5,0

Der Mittelwert alle sagt allerdings nichts aus über die Streuung um diesen Mittelwert. Dies ist ein zweites wichtiges Kriterium, denn sie sagt etwas über den Messfehler aus. So ergibt die Reihe 1.0, 5.0 und 9.0 auch den Mittelwert 5.0, in diesem Falle sind aber die Abweichungen maximal 4, beim ersten Beispiel war es nur 0,6. Ein Maß für das Streuen um den Mittelwert ist die Varianz:

Empirische Varianz: $$s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - x_m)^2$$

Aus dieser kann dann die Standardabweichung berechnet werden.

Standardabweichung: $$\sigma = \sqrt{s^2}$$

In einem Bereich um 1 Standardabweichung um den Mittelwert liegen 68.3 % der Messwerte. Bei zwei Standardabweichungen sind es schon 95.4 % der Mittelwerte und in plus/minus drei Standardabweichungen sind es 99.7 % der Messwerte.

Hintergrundinformationen finden sie unter:

http://de.wikipedia.org/wiki/Standardabweichung

und

http://de.wikipedia.org/wiki/Empirische_Varianz

Erstellen sie eine grafische Anwendung in der die Anwender an der Maschine die Sollangabe eines Teils, die Ist-Angaben einer beliebigen Anzahl von Proben eingeben kann, sowie die maximal zulässige Abweichung und berechnen sie auf dieser Basis die Standardabweichung und geben sie einen Hinweis auf das Wechseln des Verschleißteiles. Beim Auswechseln muss natürlich mit einem neuen Datensatz gearbeitet werden. Um Ausschuss zu vermeiden, wird als Auswechselkriterium die dreifache Standardabweichung genommen.