Arbeitsmaterial Delphi

Gymnasium „Geschwister Scholl“

1

Nur für den internen Gebrauch!

H. Pleske Fachbereichsleiter Informatik

Inhalt

Gymnasium „Geschwister Scholl“ 1
Erster Eindruck 4
Das Formular (Ausgabenfenster) 6
Der Quelltexteditor 6
Speichern unter Delphi 7
Öffnen eines Delphiprojektes 9
Das Formular 10
AutoScroll 10
Ereignisse 11
OnCreate 11
OnActivate 11
Variable in Delphi- Eine kurze Auswahl 15
Standardfunktionen zur Umwandlung von numerischen Daten in Zeichenketten 16
Flussdiagramme und Struktogramme 17
Schritte bei der Entwicklung eines Programms 18
Algorithmen und Programme 20
Syntaxregeln 21
Wörter, Zeichen, Bezeichner 22
Übersicht der wichtigsten Komponenten 23
Unser erstes Delphiprogramm 25
Struktogramm für diese Aufgabe 26
Das Programm Rechteck! 27
Zyklen (Schleifen) 29
Über das Wurzelziehen 30
Vorüberlegungen 31
Arbeit mit Meldungen in Delphi! 34
MUSTER ZU PEN 37
MUSTER ZU BRUSH 37
FIGUREN 39
Weitere Grafikelemente in Delphi 40
Verzweigungen 42
    if-else 42
    case-Verzweigung 43
Komplexe Datentypen 45
    Mengentypen 45
    Arrays 45

Dynamische Arrays 46
Mehrdimensionale Arrays 47
Records 47
Prozeduren und Funktionen 49
Deklaration von Funktionen und Prozeduren 51
Routinen für die Zeichenketten-Bearbeitung 52
Darstellung von Gleitkommazahlen 53
Beispiele für die Behandlung von Datum und Uhrzeit 54
Arithmetische Funktionen für reelle Zahlen 55
Beispiele für das Rechnen mit ganzen Zahlen 56
Beispiele für die Behandlung von aufzählbaren Typen 57
Arbeit mit Stringgrid (Ausgabe in Tabellen) 58
Eingabekontrolle 60
Listbox- Komponente 61
Beispielprogramm 62
Delphi- Befehle 63

Fon ist die größte WLAN Community der Welt. Jeder ist eingeladen, mitzumachen und weltweit kostenlosen WLAN Zugang zu bekommen. Alles hat mit einer einfachen Idee begonnen: Überall auf der ganzen Welt einheitlichen WLAN Zugang haben- dank der vielen Mitglieder der FON Community. Mitmachen ist ganz einfach. Du musst nur Breitbandanschluss haben, dich kostenlos bei www.fon.com registrieren, La Fonera bestellen und bei dir zu Hause anschließen und anmelden.

Werde Teil der FON Community und mach mit, die ganze Welt mit WLAN zu versorgen!

Erster Eindruck

Delphihauptfenster

Symbolleiste

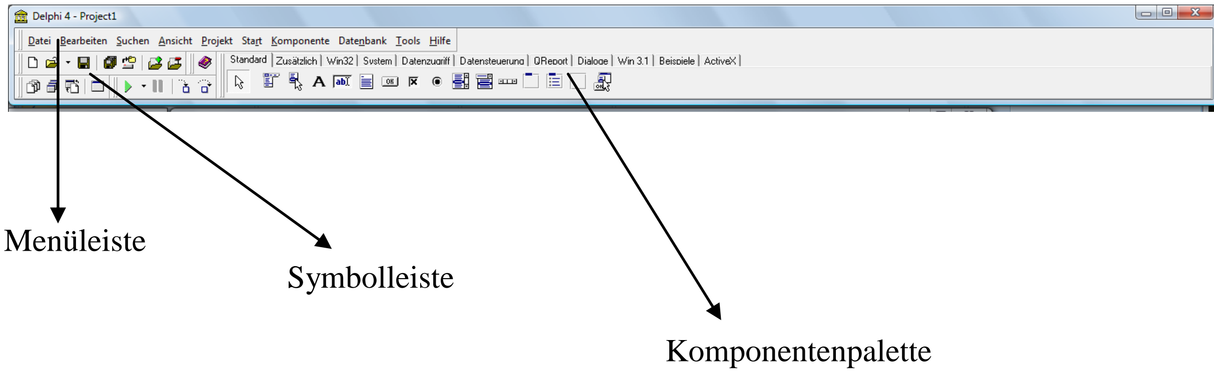
Komponentenpa

Objektinspektor

Quelltexteditor

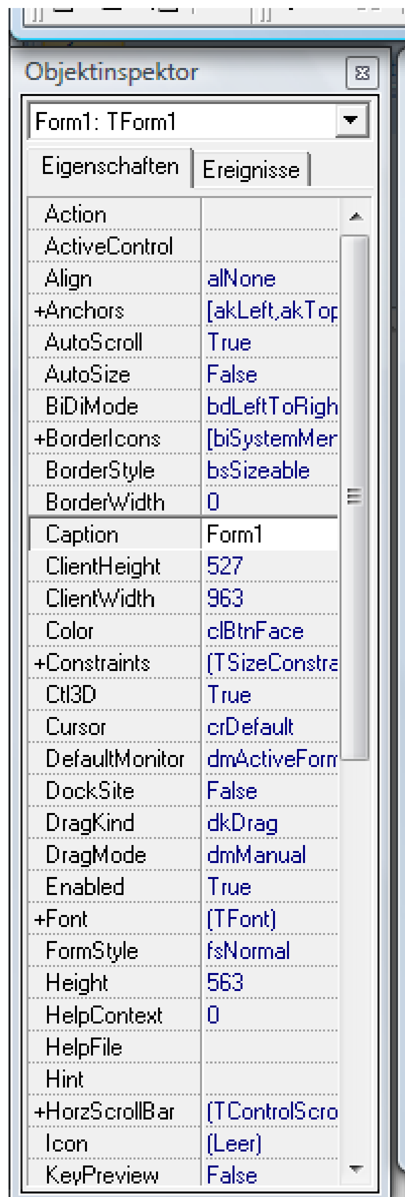
Formular

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, For

type
  TForm1 = class(TForm)
  private
    { Private-Deklaratione
  public
    { Public-Deklarationer
  end;

var
  Form1: TForm1;

implementation
```



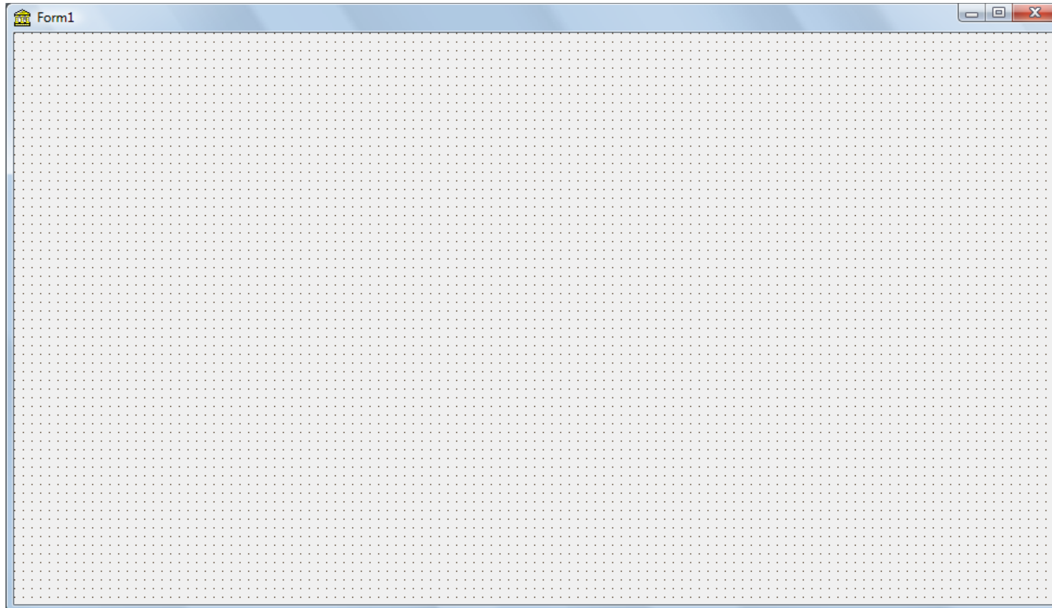
5

Objektinspektor:



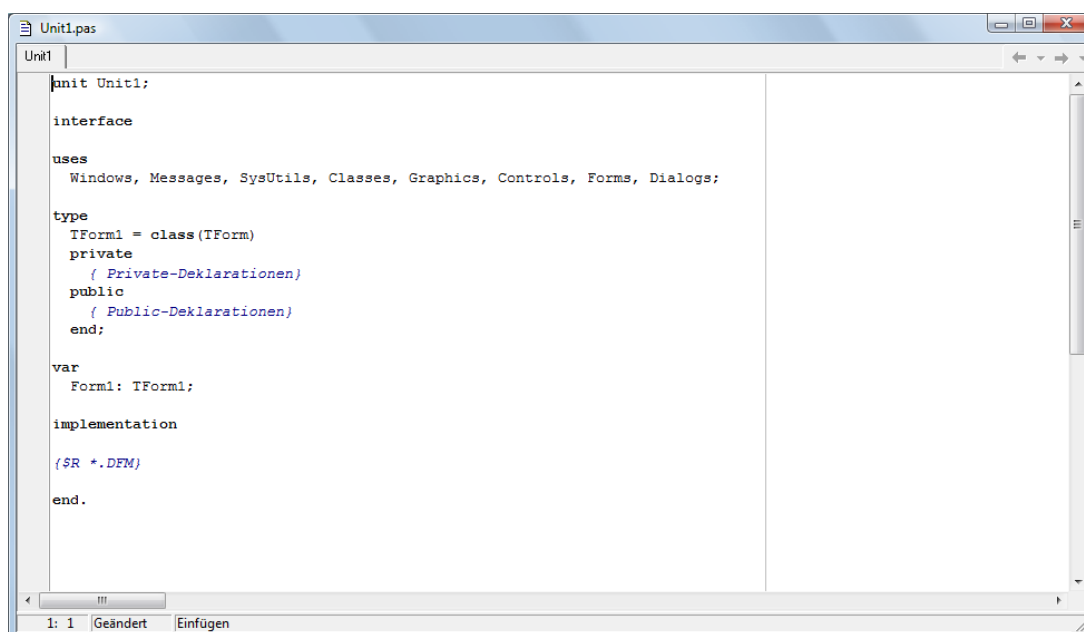
Das Formular (Ausgabenfenster)

Das Formular ist das spätere Anzeigefenster des Programmes. Auf diesem werden alle Komponenten platziert. Diese können dann ausgerichtet und verschoben werden.



6

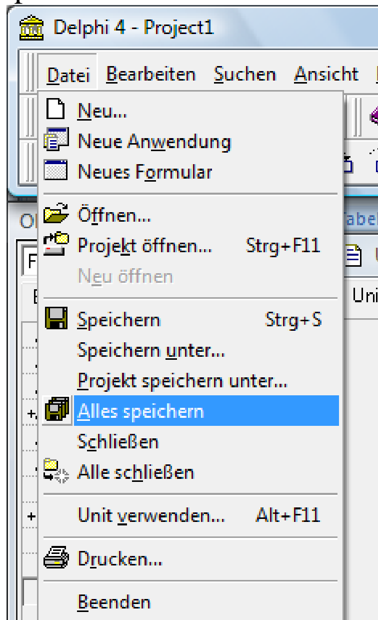
Der Quelltexteditor

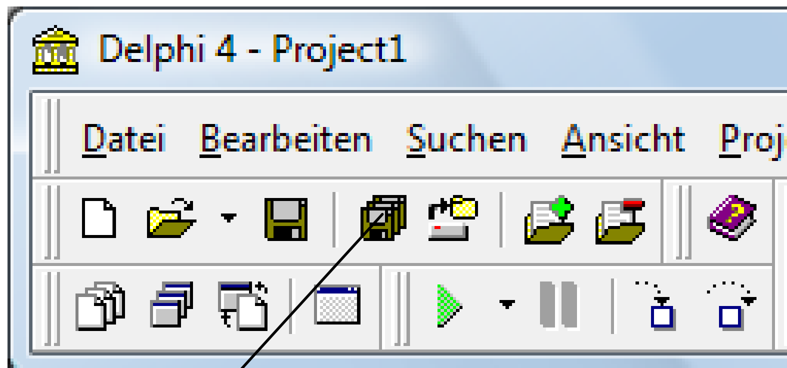


Hier geben wir unser eigenes Delphiprogramm ein.

Speichern unter Delphi

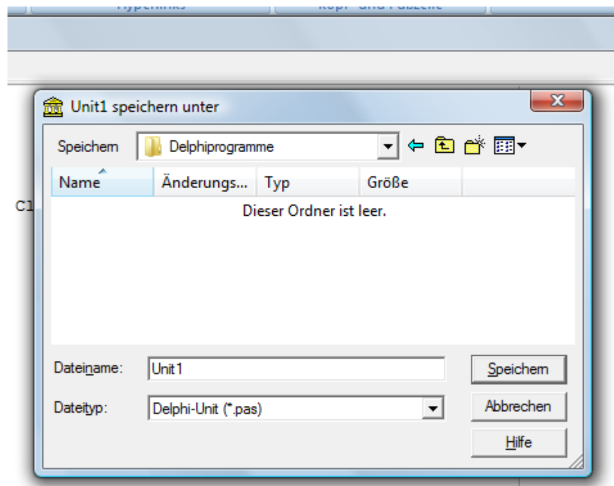
Verwende immer das als erstes den Menüpunkt alles speichern oder das Symbol für alles speichern!



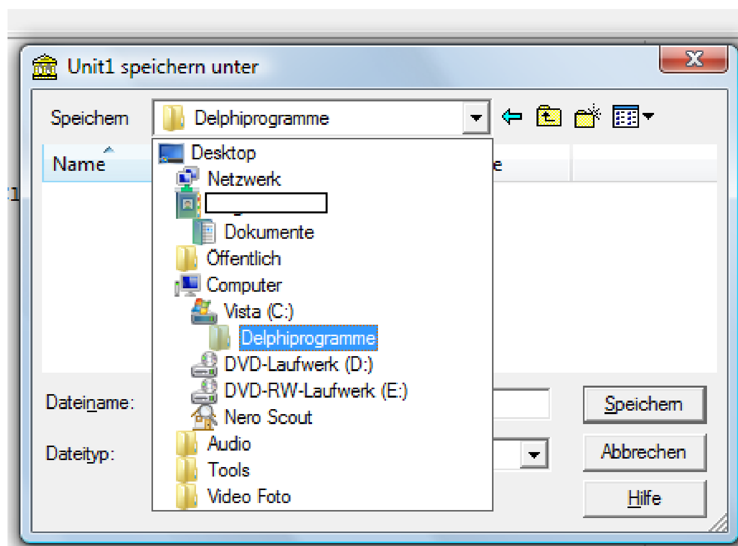


Alles speichern

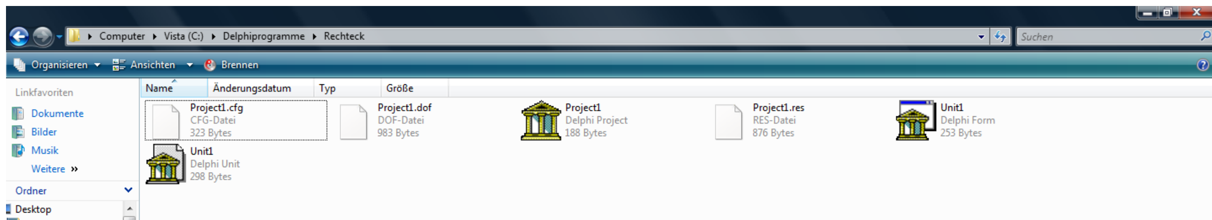
Anschließend erscheint folgendes Menü. Dieses Menü zeit je nach Installation von Delphi etwa folgendes Aussehen. Dabei weicht das angegeben Verzeichnis eventuell ab.



Anschließend klicke auf das schwarze Dreieck am oberen Rand, damit öffnet sich das Menüfenster, in dem die Baumstruktur des Rechners sichtbar wird:



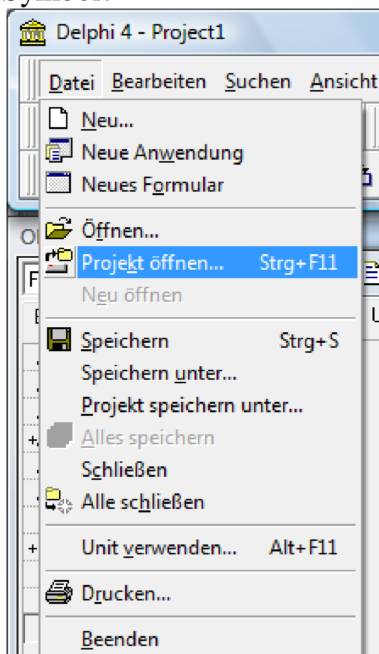
Bewege dich an den Ort, an dem dein Projekt gespeichert werden soll. Dort erstelle für jedes Delphiprogramm einen neuen Ordner. Gib diesem Ordner einen sinnvollen Namen. Dort speichere dein Programm.
Beachte dabei, das eine Unit1.pas gespeichert wird und anschließend noch eine zweite Datei (Projekt1.dpr). Es müssen beide Dateien gespeichert werden! Wenn der entsprechende Ordnerinhalt angeschaut wird erscheint folgender Inhalt:



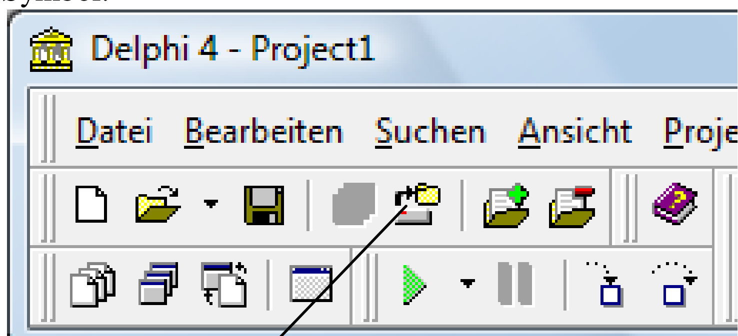
Delphi erzeugt im Hintergrund eine Vielzahl von Dateien. Nach dem ersten Start erscheinen noch weitere Dateien in diesem Ordner. Auf die verschiedenen Dateien möchte ich an dieser Stelle nicht sonderlich eingehen.

Öffnen eines Delphiprojektes

Zum Öffnen verwende immer den Menüpunkt Datei/ Projekt öffnen oder das entsprechende Symbol!



Symbol:



Projekt öffnen

Das Formular

Eigenschaften

Eigenschaft	Datentyp	Erklärung	Default
ActiveControl	TWinControl	... zeigt an, welche Komponente den Fokus hat	Keine
AutoScroll	Boolean	... legt fest, ob Bildlaufleisten angezeigt werden, wenn Formular verkleinert wird	True
BorderStyle	TFormBorderStyle	... legt Art des Rahmens fest bsDialog ➔ nicht größenveränderlich; einfaches Dialogfenster bsSingle ➔ nicht größenveränderlich; einfache Rahmenlinie bsNone ➔ nicht größenveränderlich; keine sichtbare Rahmenlinie; kein Schalter Symbol und Vollbild, kein Steuermenü bsSizable ➔ größenveränderlicher Standardrahmen	BsSizeable
Caption	String	... beinhaltet den in der Titelleiste angezeigten Text	wie Name
ClientHeight ClientWidth	Integer	... entspricht der Höhe bzw. Breite des Clientbereiches in Pixeln	wie im Editor dargestellt
Color	TColor	... legt Hintergrundfarbe fest	ClBtnFace
Ctl3D	Boolean	... legt drei- oder zweidimensionales Aussehen fest	True
Cursor	TCursor	... bestimmt das Aussehen des Mauszeigers, wenn er sich über dem Formular befindet	CrDefault
Enabled	Boolean	... legt fest, ob Formular auf Maus-, Tastatur- und Timer- Ereignisse reagiert	True
+Font	tFont	... bestimmt die Textausgaben ➔ Objekt für nachfolgende Text-eigenschaften (Color, Name, Size, Style)	System-schrift
FormStyle	TFormStyle	... bestimmt Formulartyp (normal, MDI- Rahmenfenster, MDI-Kindfenster, immer oben)	fsNormal
Height Width	Integer	... entspricht der Gesamthöhe bzw. –breite des Formulars in Pixeln	wie im Editor dargestellt

10

Left Top	Integer	... entspricht dem Abstand der linken bzw. oberen Kante des Formulars zum Bildschirmrand in Pixeln	-1
WindowState	TWindowState	... legt Größe des Formulars beim Programmstart fest (normal, maximal, Ikone)	wsNormal

Ereignisse

Formularereignisse

Ereignis	... tritt ein wenn
OnCreate	... das Formular zum ersten Mal erzeugt wird
OnActivate	... das Formular aktiviert wird (in den Vordergrund kommt)
OnClose	... das Formular geschlossen wird
OnCloseQuery	... das Formular nach einer Abfrage geschlossen wird.
OnPaint	... vorher verdecktes Formular neu gezeichnet werden muß
OnResize	... die Größe des Formulars geändert wird.
OnShow	... bevor das Formular sichtbar wird.

OnHide	... das Formular unsichtbar gemacht wurde.
OnDeactivate	... zu einer anderen Windows- Applikation gewechselt wird.
OnDestroy	... wenn das Formular (oder Hauptfenster) entfernt wird.

Operator	Operandentyp	Ergebnis typ	Beispiel	Beschreibung

+	Real
	Integer
	Integer/Real
	Real/Integer
		Real
		Integer
		Real
		Real
			`rZahl:=10.1+10.2;`
			`iZahl:=3+4;`
			`rZahl:=4+3.1;`
			`rZahl:=5.432+7;`
				Es wird bei numerischen Datentypen die üblich Addition durchgeführt.

+	String/Char	String String	`sWort:='Hal'+'lo;` `sWort:= 'S'+'O';`	Bei Zeichenketten wird eine Verkettung durchgeführt, indem String- oder Char-Datentypen aneinander gehängt werden.

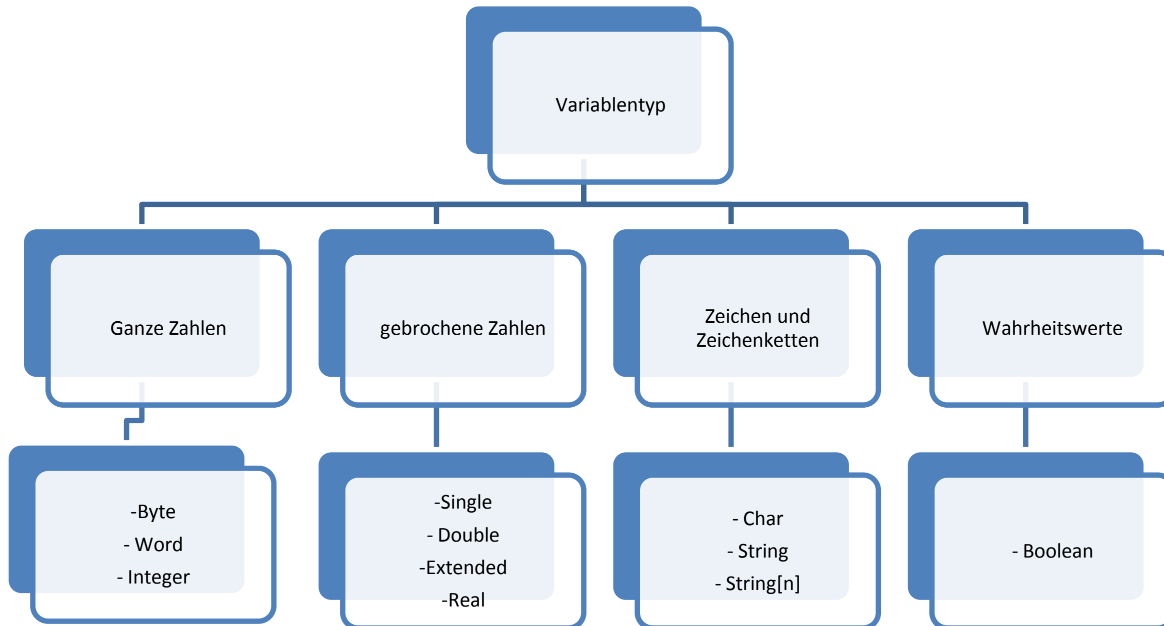
-	Integer
	Real
	Integer
	Real
		Integer
		Real
		Real
		Integer
			`iZahl:=10-2;`
			`rZahl:=2.1-0.89;`
			`rZahl:=5+5.64;`
			`rZahl:=4+67.86;`
				Bei numerischen Datentypen wird die übliche Subtraktion durchgeführt.

*	Integer
	Real
	Integer
	Real
		Integer
		Real
		Real
		Integer
			`iZahl:=10*12;`
			`rZahl:=1.1*1.1;`
			`rZahl:=4*16.43;`
			`rZahl:=34.3*21;`
				Bei numerischen Datentypen wird die übliche Multiplikation durchgeführt.

/	Integer/Integer Integer/Real Real/Real	Real	`rZahl:=10/12;` `rZahl:=10.1/10.2;`	Bei numerischen Datentypen wird die übliche Division durchgeführt. Auch wenn diese Division mit Integer-Datentypen durchgeführt wird, wird das Ergebnis intern zu einem Real-Typ umgewandelt.
div	Integer	Integer	`iZahl:=45 div 13;` ***Ergebnis: iZahl=3***	Bei dieser Division von Integer-Datentypen werden die Nachkommastellen im Ergebnis abgeschnitten.
mod	Integer	Integer	`iZahl:=45 mod 13;` *Ergebnis: iZahl=6*	Über diesen Operator kann man den Rest bei einer Integer-Division bestimmen.

Variablentypen

Hier ein kleiner Überblick über einige wichtige Variablentypen.



Variable in Delphi- Eine kurze Auswahl

1. Speichern von ganzen positiven Zahlen

Variablentyp	Minimum	Maximum	Speicherbedarf
Byte	0	255	1 Byte
Word	0	65 535	2 Byte

2. Speichern von ganzen Zahlen

Variablentyp	Minimum	Maximum	Speicherbedarf
ShortInt	-128	127	1 Byte
Integer (Delphi 1)	- 32 768	32 767	2 Byte
Integer (ab Delphi 2)	2 147 483 648	2 147 483 647	4 Byte
LongInt	2 147 483 648	2 147 483 647	4 Byte

3. Reelle Zahlen

Variablentyp	Minimum	Maximum	Speicherbedarf
Real			
Single			
Double			
Extended			

4. Variablen zum Speichern von einzelnen Zeichen

Variablentyp	Minimum	Maximum	Speicherbedarf
Char			

5. Variablen zum Speichern von Zeichenketten

Variablentyp	Minimum	Maximum	Speicherbedarf
String			
ShortString			

6. Variablen zum Speichern von Wahrheitswerten

Variablentyp	Minimum	Maximum	Speicherbedarf
Boolean			

Standardfunktionen zur Umwandlung von numerischen Daten in Zeichenketten

Funktion	Bedeutung	Beispiel / Bemerkung
StrToInt	Versucht, eine Zeichenfolge in einen Integerwert umzuwandeln.	x := StrToInt(E_Eingabe.Text);
IntToStr	Wandelt einen Integerwert in einen String um.	Label1.Caption := 'Das Ergebnis ist ' + IntToStr(x) + '.';
StrToFloat	Versucht, eine Zeichenfolge in einen Realwert umzuwandeln.	x := StrToFloat(Edit1.Text);
FloatToStr	Wandelt einen Realwert in einen String um.	Label1.Caption := 'Das Ergebnis ist ' + FloatToStr(x) + '.';
IntToHex	Wandelt einen Integerwert in einen String um, wobei das Zahlenformat im Hexadezimalsystem benutzt wird.	Label1.Caption := 'Die Zahl schreibt sich ' + IntToHex(x) + '.';
StrToIntDef	Wandelt eine Zeichenfolge in eine Ganzzahl um. Lässt sich kein Interwert konstruieren, wird stattdessen ein Standardwert zurückgegeben.	x := StrToFloat(E_Eingabe.Text);

Flussdiagramme und Struktogramme

Struktogramme bzw. Flussdiagramme machen die logischen Strukturen eines Algorithmus grafisch anschaulich. Sie sind keine Schikane des Informatiklehrers, sondern eine notwendige Voraussetzung der Programmdokumentation zur Entwicklung und Wartung eines Programms. Man verwendet dafür genormte Sinnbilder nach DIN 66001 (Programmablaufplan) oder auch DIN 66261 (Struktogramme nach Nassi und Shneiderman).

Sinnbilder für Programmablaufpläne	
Sinnbild	**Benennung**
	Operation allgemein
	Unterprogrammaufruf
	Ein- bzw Ausgabe
	Verzweigung
	Übergangsstelle
	Grenzstelle
	Ablauflinien
	Schleifenbegrenzung für zählergesteuerte Wiederholungen

Elemente von Struktogrammen	
Sinnbild	**Benennung**
Anweisung 1 Anweisung 2 Anweisung 3	Folgeblock für Wertzuweisungen, Rechenoperationen und Bildschirmbefehle
Anweisung 1 Anweisung 1	Eingabeanweisung (nicht genormt) Ausgabeanweisung (nicht genormt)
Anweisung 1 Anweisung 2 wiederhole bis ...	Wiederholungsstruktur mit Endebedingung
wiederhole solange Anweisung 1 Anweisung 2	Wiederholungsstruktur mit Anfangsbedingung
Bedingung nein / ja Anweisung	Verzweigungsblock a) einseitig b) zweiseitig
Bedingung Fall 1 / 2 / 3 / 4..	Verzweigungsblock mehrfach
Aufruf U-Prg	Aufruf einer Subroutine a) Funktion b) Prozedur

Schritte bei der Entwicklung eines Programms

Ganz gleich mit welcher Programmiersprache man für den Computer ein Programm entwickeln will - die folgenden Schritte sind in jedem Fall zweckmäßig, wenn nicht sogar notwendig:

1. **Problemanalyse**

   Als erstes muss die Problemstellung geklärt werden. Es muss untersuchucht werden, ob das Problem algorithmierbar ist. Das Problem muss genau umrissen werden. Das Ziel, die Anfangsbedingungen und die Grenzen sollten vereinbart werden. Möglichst alle denkbaren Situationen müssen vorher analysiert und geplant werden. Dann werden die Problemdaten erfasst und die benötigten Werkzeuge bereit gestellt.

2. **Datenanalyse**

   Es muss festgelegt werden, welche Daten ausgegeben werden sollen und welche Daten dafür einzugeben sind. Dabei ist zu klären, welche Datentypen das Programm also benutzen soll.

3. **Programm-Entwurf (Algorithmus)**

   Es ist eine Programmstruktur im Rahmen der durch die Programmierumgebung vorgegebenen Bedingungen zu erarbeiten.
   Ein Algorithmus ist die Folge von Schritten, die dazu dienen, eine Aufgabe zu lösen. Der Computer versteht weniger als ein Vorschulkind. Deshalb muss das Problem in winzig kleine logisch aufeinander folgende Schritte zerlegt werden. Es muss vorher schon klar sein, was unter welcher Bedingung wann getan werden soll, um das Ziel zu erreichen.
   Es empfiehlt sich, den Algorithmus visuell anschaulich zu gestalten. Dazu kann man Pseudocode, einen Programmablaufplan (Flussdiagramm) nach DIN 66001 oder ein Struktogramm nach DIN 66261 verwenden.

4. **Programm-Erstellung**

   Programme sind eine exakt definierte, in einer bestimmten Syntax geschriebene Sammlung von Arbeitsschritten, die der Computer durchführen soll. Zur Herstellung von Programmen werden diese i.d.R. in einer Programmiersprache geschrieben (z. B. BASIC, PASCAL, C oder Delphi), die aus einer für den Menschen verständlichen Sammlung von Befehlen und Anweisungen besteht.
   Das Programms wird in einer Form niedergeschrieben, die auf dem Computer abgearbeitet werden kann.
   Dazu wird es in irgendeiner Form über einen Editor Zeichen für Zeichen und Zeile für Zeile geschrieben. Wie sich eine gotische Kathedrale letztlich aus vielen einzelnen Steinen zusammen setzt, so besteht auch ein Programm aus einzelnen Zeichen, wobei allerdings schon ein einziges fehlendes oder falsch gesetztes Zeichen verheerende Folgen haben kann, während die Kathedrale auch mit mehreren fehlenden Steinen noch brauchbar und wirkungsvoll bleibt.
   Um diesen Programm-Code in die für den Computer allein verständliche Maschinensprache zu überführen, gibt es zwei Möglichkeiten: Entweder wird er während der Abarbeitung der Schritte durch eine Software Zeile für Zeile interpretiert (übersetzt) oder er wird von der Programmier-Software komplett kompiliert.
   Der beim Kompilieren resultierende Maschinencode ist in einer oder mehrerer Dateien zum endgültigen Programm zusammengefasst und i.d.R. über seinen Namen aufrufbar.
   Ein so aufgerufenes Programm wird zur Ausführung vom Datenträger in den Arbeitsspeicher des PCs geladen und abschließend vom Prozessor abgearbeitet.

5. **Programm-Test**

Das Programm muss unter reproduzierbaren Bedingungen mit vorher festgelegten Daten überprüft werden, die möglichst viele Programmabläufe erfassen.
Das Programm wird in zwei Phasen getestet. Zuerst muss die Syntax überprüft werden, damit es überhaupt läuft. Diese Überprüfung führt bei Delphi der Compiler aus. In der zweiten Testphase wird die Logik überprüft. Logische Fehler können gefährliche Konsequenzen haben.
Das Programm sollte auch durch den DAU (**D**ümmster **a**nzunehmender **U**ser) bedient und nicht zum Absturz gebracht werden können.

6. **Programm-Dokumentation**

Es ist sehr zweckmäßig, die Struktur, die Leistungsfähigkeit, die Handhabungsvorschriften und das Vorgehen bei möglichen Fehlermeldungen zu beschreiben.
Abschließend wird notiert, was das Programm leistet und wie es bedient wird. Zur Dokumentation gehört neben dem Flussdiagramm bzw. Struktogramm auch das Programmlisting.

Algorithmen und Programme

Schnell ist mit Delphi eine Windows-Anwendung zusammengeklickt - aber für richtige Programme muss man doch das Programmieren mit Objekt-Pascal lernen. Dazu gehören in erster Linie die Syntax, die Erstellung von Algorithmen und das Umwandeln von Algorithmen in Programme.

Die wichtigsten Syntaxregeln sind:

- Groß- bzw. Kleinschreibung spielt bei der Windows-Version von Delphi keine Rolle (bei Kylix für Linux allerdings doch). Deshalb kann man zweckmäßigerweise Großbuchstaben innerhalb von Variablen- oder Funktionsnamen zur Veranschaulichung benutzen. (z. B. IntToStr(i))
- Leerräume und neue Zeilen (White-Spaces) werden ignoriert.
- Das Semikolon trennt in Pascal zwei Anweisungen. An das Ende eines Befehls gehört immer ein Semikolon.
- Kommentare im Programmcode stehen zwischen geschweiften Klammern { Kommentar } bzw. zwischen der Kombination aus runder Klammer und Sternchen (* Kommentar *). Sie werden vom Delphi-Compiler ignoriert und dienen der Übersichtlichkeit des Quellcodes. Kommentare können auch über mehrere Zeilen gehen.
- Wenn man am Ende einer Programmzeile // schreibt, wird der restliche Teil zum Kommentar, also auskommentiert.

Ein **Algorithmus** ist eine Verarbeitungsvorschrift, die aus einer endlichen Folge von eindeutig ausführbaren Anweisungen besteht, mit der man eine Vielzahl gleichartiger Aufgaben lösen kann.

Endlichkeit

Ein Algorithmus besteht aus endlich vielen Anweisungen von jeweils endlicher Länge

Eindeutigkeit

Die Reihenfolge der Abarbeitung der Anweisungen unterliegt nicht der Willkür des Ausführenden. Bei gleichen Bedingungen und gleichen Eingangsgrößen muss man stets zu gleichen Ergebnissen kommen.

Ausführbarkeit

Jede Anweisung muss vom Ausführenden verstanden werden und ausgeführt werden können, d.h. ein Algorithmus ist nur für den Ausführenden ein Algorithmus, der ihn versteht und ausführen kann.

Allgemeingültigkeit

Ein Algorithmus muss auf alle Aufgaben gleichen Typs anwendbar sein. Lösbarkeit oder Nichtlösbarkeit jeder Aufgabe müssen eindeutig sein.

Ein Algorithmus gibt an, wie Eingabegrößen in Ausgabegrößen umgewandelt werden.

Als **Programm** bezeichnet man einen Algorithmus, der in einer dem Computer verständlichen Sprache formuliert ist und von diesem ausgeführt werden kann.

Syntaxregeln

Object-Pascal ist sehr tolerant gegenüber Schreibweisen und sogar -fehlern. Was es nicht akzeptiert oder "errät", wird beim Compilieren erkannt und angezeigt. Man versteht aber manche Fehlermeldung nur, wenn man die wichtigsten Syntaxregeln kennt.

- Groß- bzw. Kleinschreibung spielt (bei der Windows-Version) keine Rolle.
  (Das kann man zur besseren Übersichtlichkeit geschickt ausnutzen.)

- White-Spaces (Leerräume, Tabulatoren und neue Zeilen) werden ignoriert.
  (Auch dadurch kann man den Quellcode strukturieren.)

- Zwei Anweisungen - komplette Befehle sozusagen - werden immer durch ein Semikolon getrennt.
  (Man sollte an das Ende eines Befehls immer ein Semikolon setzen, auch wenn das in Ausnahmefällen nicht immer notwendig ist.)

- Reservierte Wörter, die von Object-Pascal als Befehle erkannt wurden, werden fett dargestellt.
  (Sie dürfen nicht für andere Zwecke verwendet werden.)

- Kommentare im Programmcode stehen zwischen geschweiften Klammern: {Kommentar} oder zwischen (* *): (*Kommentar*).
  Kommentare lassen sich auch durch zwei Divisionszeichen einleiten: //Kommentar
  Ein solcher Kommentar reicht bis zum Zeilenende.
  (Kommentare dienen dem Programmierer zum Verständnis des Codes auch zu späteren Zeiten.)

- Innerhalb eines Programms fasst man mit **begin ... end** Anweisungen zu einem Block zusammen. Ein solcher Block wird wie eine einzelne Anweisung betrachtet, daher trennt man mehrere Blöcke bzw. Anweisungen durch Semikolons.
  (Das Programm selbst beginnt mit **begin** und endet mit **end**, auf das letzte **end** folgt jedoch kein Semikolon, sondern ein Punkt.)

- Es ist zweckmäßig (aber nicht notwendig) die Namen der Komponenten, d. h. ihre Eigenschaft "Name", durch einen anschaulichen Begriff zu ersetzen. Dann ist allerdings auch zu empfehlen, sich an die ungarische Notation zu halten (wie z. B. auch bei Visual Basic).
  Die ungarische Notation (Hungarian Notation) besteht aus detaillierten Richtlinien zur Benennung von Variablen. Diese Notation hat in der Programmierung in C, besonders in der Windows-Welt, weite Verbreitung gefunden. Sie wird "ungarisch" genannt, weil der Vater dieser Notation, Charles Simonyi, ein gebürtiger Ungar ist und die Namen auf den ersten Blick tatsächlich ein wenig fremdländisch wirken.

Wörter, Zeichen, Bezeichner

- Reservierte Wörter
  Reservierte Wörter haben für den Delphi-Compiler jeweils eine ganz bestimmte Bedeutung - sie sind für einen bestimmten Zweck reserviert. Es sind Anweisungen der Sprache Pascal.
  Reservierte Wörter sind u. a.: **and, array, as, asm, begin, case, class, const, constructor, destructor, div, do, downto, else, end, except, exports, file, finalization, finally, for, function, goto, if, implementation, in, inherited, initialization, inline, interface, is, label, library, mod, nil, not, object, of, on, or, packed, procedure, program, property, raise, record, repeat, set, shl, string, then, to, threadvar, try, type, unit, until, uses, var, while, with, xor**.

- Standardanweisungen
  Standardanweisungen werden nicht ganz so streng behandelt wie reservierte Wörter. Aber es ist fast nie sinnvoll, sie neu zu definieren.
  Standardanweisungen sind z. B. **absolute, abstract, assembler, at, automated, cdecl, default, dynamic, export, external, far, forward, index, interrupt, message, name, near, nodefault, override, private, protected, public, published, read, register, resident, stdcall, stored, virtual, write**.

- Reservierte Zeichenfolgen
  Bestimmte Sonderzeichen haben für den Delphi-Compiler natürlich auch einen ganz bestimmten Zweck (z. B. Semikolon und Punkt, die wir schon beschrieben haben).
  Folgende Zeichen und Zeichenkombinationen sind reserviert: **+ - * / [:= $ (. . , ; :] = # .) < > <> <= >= { (* ^ ( ) .. @ } *)**.

- Bezeichner
  Konstanten, Variablen, Typen, Funktionen, Programme und Units können wir mit Namen versehen, sogenannten Bezeichnern. Dann reserviert Delphi gewissermaßen eine Schublade mit einer bestimmten Größe und Art für sie, beschriftet diese mit dem Bezeichner, und das Programm kann diese Schublade beliebig füllen oder leeren. Dazu braucht nur der Bezeichner aufgerufen zu werden, um Zugang zum jeweiligen Inhalt zu erlangen.
  Bezeichner dürfen maximal 63 ASCII-Zeichen (also z. B. keine Umlaute) lang sein.
  Ein Bezeichner muss mit einem Buchstaben oder dem Unterstrich (_) beginnen.
  Ein Bezeichner kann kein Leerzeichen oder andern White-Space enthalten.
  Als Bezeichner einer Variable kann nicht noch einmal der gleiche Name vergeben werden, wie schon z. B. die Unit heißt.

Übersicht der wichtigsten Komponenten

Typ	Aufgabe	Benutzung / Beispiele
Form	Formular, späteres Programmfenster	Im Objektinspektor Eigenschaft **Caption** (Beschriftung) festlegen oder zur Laufzeit: `Form1.Caption := ´Titel der Anwendung´;`
Label	Anzeige einer Textzeile auf dem Formular	Im Objektinspektor Eigenschaft **Caption** (Beschriftung) festlegen oder zur Laufzeit: `Label1.Caption := ´Hallo!´;`

Edit	Anzeige eines Textes in einem Eingabefeld mit Möglichkeit zum Editieren.	Im Objektinspektor Eigenschaft **Text** festlegen oder über Wertzuweisung Eigenschaft **Text** verändern oder verwenden:
		`Edit1.Text := ´Hallo Welt!´;`
		`s := Edit1.Text; { s: String }`

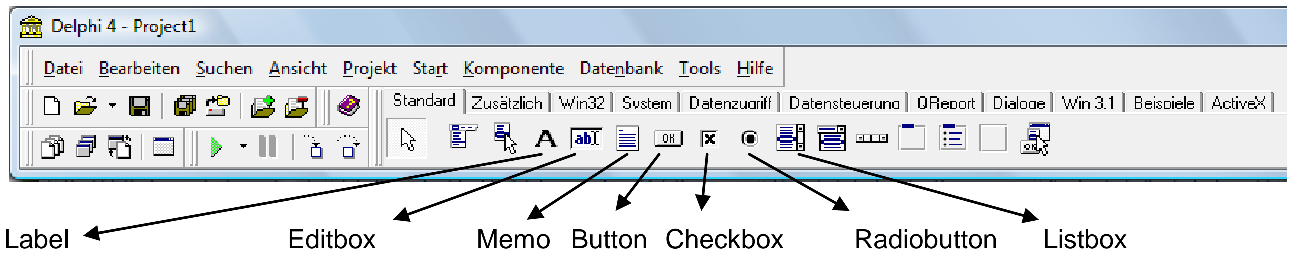
Button	Schaltknopf zum "Anklicken", um eine Aktion auszulösen.	Im Objektinspektor Eigenschaft **Caption** festlegen. Bei der Formularentwicklung wird durch einen Doppelklick auf den Button die Ereignisprozedur Button1Click geöffnet.

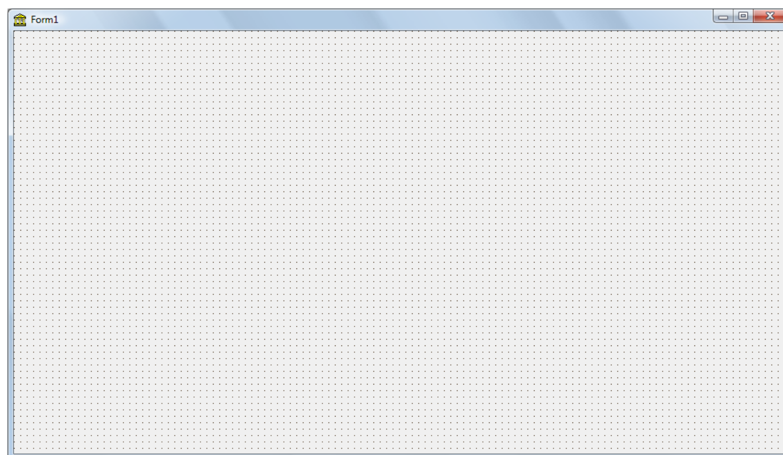
CheckBox	Feld zum Ankreuzen per Mausklick	Im Objektinspektor Eigenschaft **Caption** festlegen und mit der Eigenschaft **Checked** evtl. Kreuzchen vorgeben.
		`CheckBox1.Checked then ... ;`
		`CheckBox1.Checked := False;`

RadioBox	Optionsfelder, von denen immer nur eines aktiv sein kann	Im Objektinspektor Eigenschaft **Caption** festlegen und mit der Eigenschaft **Checked** evtl. Auswahl vorgeben.
		`RadioBox1.Checked then ... ;`
		`RadioBox1.Checked := False;`

ListBox	Anzeige mehrerer Textzeilen.
	
	Enthält als Objekt **Items** eine Stringliste mit dynamischer Länge.
		Eigenschaften :
		`ListBox1.Items.Count` - (Anzahl der Einträge),
		`ListBox1.ItemIndex` - (Nr. der aktuellen Zeile -1),
		`ListBox1.Items[i]` - (i-ter Eintrag),
		`ListBox1.Items.Sorted`
		
		Methoden :
		`ListBox1.Clear,`
		`ListBox1.Items.Add(´Hallo´),`
		`ListBox1.Items.Delete(i),`
		`ListBox1.Items.Insert(i,s),`
		`ListBox1.Items.Exchange(2,5),`
		`ListBox1.Items.LoadFromFile(´TEST.TXT’),`
		`ListBox1.Items.SaveToFile(´TEST.TXT´)`

Memo	Anzeige mehrerer Textzeilen mit Editiermöglichkeit.	Mit Außnahme von **ItemIndex** die gleichen Eigenschaften und Methoden wie bei ListBox.
StringGrid	Zweidimensionales Feld zur Anzeige von Strings in Tabellenform	**Eigenschaften :** `StringGrid1.Cells[s,z]` - String in Spalte s, Zeile z `StringGrid1.RowCount` - Anzahl der Zeilen `StringGrid1.ColCount` - Anzahl der Spalten `StringGrid1.FixedRows` - Anzahl der oben fixierten Zeilen `StringGrid1.FixedCols` - Anzahl der links fixierten Spalten





Formular

Unser erstes Delphiprogramm

Rechteck

Aufgabe:

Nach Eingabe der beiden Seitenlängen eines Rechtecks sollen der Flächeninhalt, der Umfang und die Länge der Diagonalen ausgegeben werden!

Vorüberlegungen:

Jeder PC arbeite nach dem EVA- Prinzip!

Eingabe	Verarbeitung	Ausgabe
• Seite a • Seite b	• Berechne Flächeninhalt • Berechne Umfang • Berechne Diagonale	• Ausgabe des Flächeninhalts • Ausgabe des Umfangs • Ausgabe der Diagonalenlänge

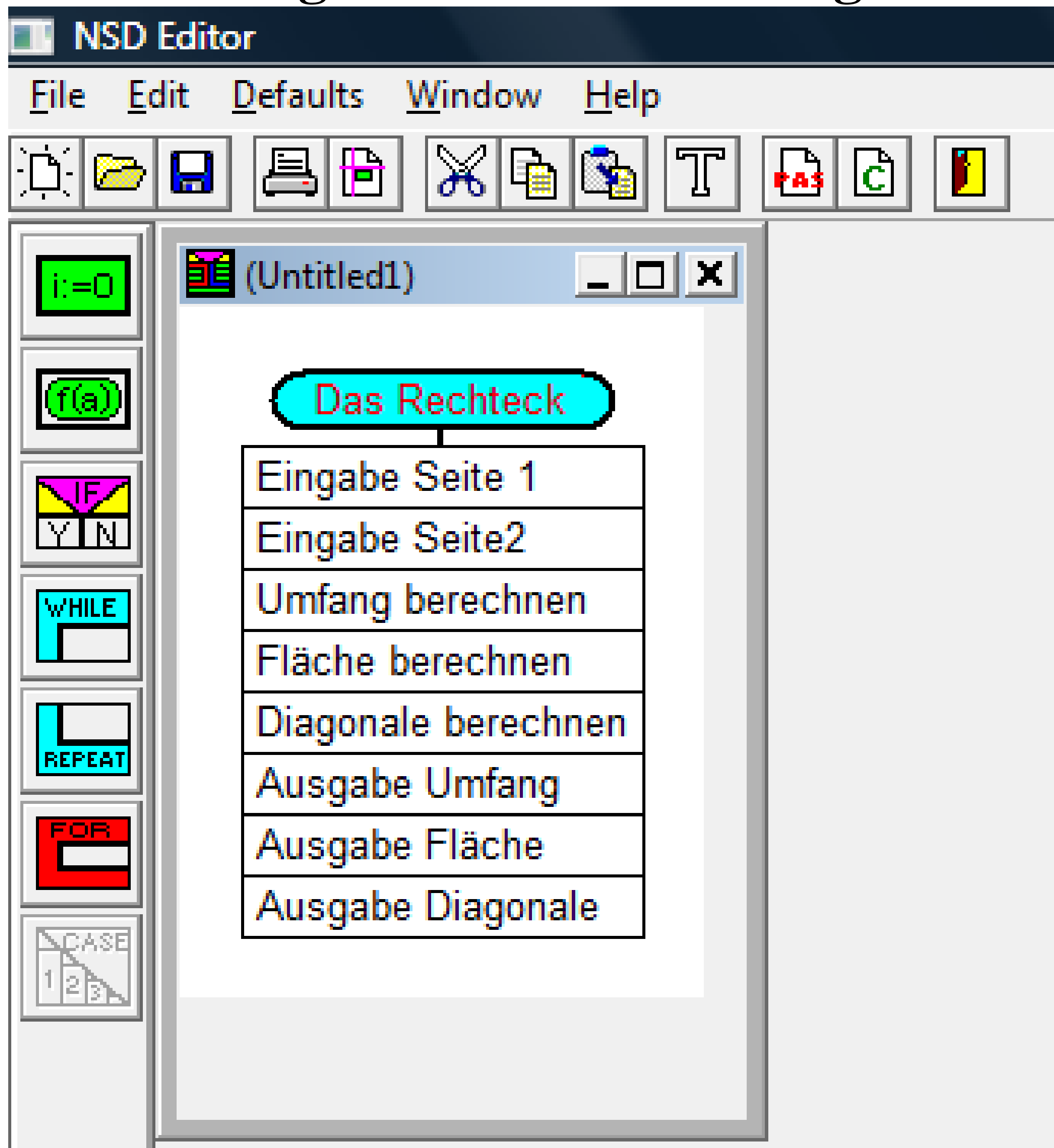
Auch das behalten wir bei der Programmierung bei!

Variablen die wir benötigen:

Seite_a; Seite_b; Flaecheninhalt; Umfang;Diagonale

Mathematisch sind alles Zahlen aus dem Bereich der reellen Zahlen. (Eine Absicherung für negative Zahlen ist hier noch nicht vorgesehen)

Struktogramm für diese Aufgabe



Vorbereitung des Formulars:

Das Programm Rechteck!

Komponenten	Berechnungen am Rechteck	Name der Komponenten
Panel	Berechnungen am Rechteck	P_Ueberschrift
Editbox	Seite a ___ cm Seite b ___ cm Fläche ___ cm² Umfang ___ cm Diagonale ___ cm	E_Seitea E_Seiteb E_Flaeche E_Umfang E_Diagonale
Label		
Button	Rechnen Beenden	B_Berechnen B_Beenden

Hinweis: Vergib möglichst sinnvolle Komponentenname

```
procedure TForm1.RechnenButtonClick(Sender: TObject);
var Seite1,Seite2,Flaeche,Umfang,Diagonale : Single;
begin
// Eingabe
  Seite1:=StrToFloat(Seite1Edit.Text);
  Seite2:=StrToFloat(Seite2Edit.Text);

// Verarbeitung
  Flaeche:=Seite1*Seite2;
  Umfang:=2*(Seite1+Seite2);
// SQR(x) ➔ √X
  Diagonale:=Sqrt(Sqr(Seite1)+Sqr(Seite2));
// Ausgabe
  FlaecheEdit.Text:=FloatToStrF(Flaeche,ffFixed,10,2);
  UmfangEdit.Text:=FloatToStrF(Umfang,ffFixed,10,2);
  DiagonaleEdit.Text:=FloatToStrF(Diagonale,ffFixed,10,2);
end;
```

Zyklen (Schleifen)

1. Zählzyklus

Die FOR- Schleife ist eine Form des abweisenden Zyklus. Sie dient dazu, festzulegen, wie oft andere Anweisungen abgearbeitet werden. Eine bestimmte Variable wird als Laufvariable definiert. Die Laufvariable wird nach jedem Durchlauf um 1 erhöht (TO) bzw. um 1 verringert (DOWNTO). Diese Laufvariable muss einen ordinalen Typ haben. Dieser wird ein bestimmter Anfangswert zugewiesen.
Die Abarbeitung der Schleife wird ausgeführt, bis die Laufvariable den Endwert erreicht bzw. überschritten (unterschritten) hat.
Bsp:


```
For i:= n downto 1 do Zahl:=Zahl*I;
```

2. Abweisende Zyklen

Die WHILE- DO- Anweisung ist ein abweisender Zyklus, da vor Eintritt in den Zykluskörper die Abbruchbedingung geprüft wird und somit der Zyklus nicht unbedingt abgearbeitet werden muss.
Bsp.:

```
While ((kreditsumme>0) and (monate<3600)) do
Begin
…..
end;
```

3. Nichtabweisender Zyklus

Die REPEAT- UNTIL- Anweisung ist ein “nichtabweisender Zyklus”. Beim nichtabweisenden Zyklus wird die Bedingung erst für das Verlassen des Zykluskörpers ausgewertet. Dieser Zyklus wird unabhängig vom Wert der Bedingung mindestens einmal durchlaufen.

Bsp.:

```
Repeat
......
until ((kreditsumme>0) and (monate<3600))
```

Über das Wurzelziehen

Die alten Babyloner benutzten folgende Formel:

$$\sqrt{a^2+b} = a + \frac{b}{2a}$$

Beispiel:

$$\sqrt{40} = \sqrt{36+4} = 6 + \frac{4}{2*6} = 6,33$$

(Ein genauer Wert ist 6,3245553)

Bessere Näherungen gehen auf *Heron von Alexandria* (100 u. Z.) zurück!
a_1 **sei ein erster Schätzwert für** $\sqrt{x}$**. Dann berechnet man einen verbesserten Schätzwert** a_2 **mit Hilfe der Formel:**

$$a_2 = \frac{1}{2}\left(a_1 + \frac{x}{a_1}\right)$$

und erforderlichenfalls weitere Verbesserungen a_{n+1} **mittels**

$$a_{n+1} = \frac{1}{2}\left(a_n + \frac{x}{a_n}\right)$$ **für n=1,2,3,...**

Dieses Verfahren wird auch Interationsverfahren genannt!

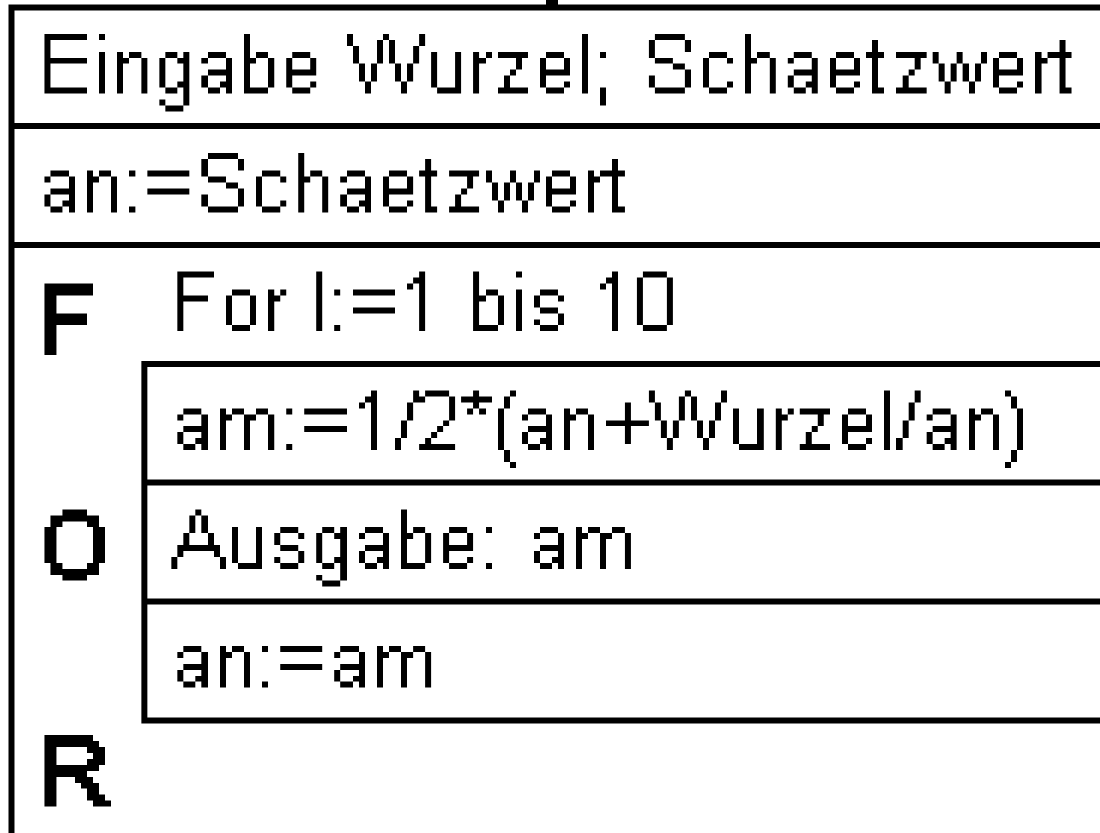
Vorüberlegungen

Struktogramm und Variablen:

Variablen:

Wurzel	Zahl aus der die Wurzel gezogen werden soll
Schaetzwert	erste Schätzung
an	n- te Näherung
am	n+1 - Näherung

Variablengrundbereiche:
➔ reelle Zahlen!; Wurzel nur positive Werte!



Der Quelltext

```
procedure TF_Haupt.B_zaehlClick(Sender: TObject);
Var Wurzel,Schaetzwert,an,am:single;
   i:Integer;
begin
// Eingabe und prüfen auf Gültigkeit
Repeat
Wurzel:=StrToFloat(E_wurzel.Text);
Schaetzwert:=StrToFloat(E_Schaetzwert.text);
If ((wurzel<=0)OR (Schaetzwert=0)) then begin
Showmessage('Wurzel muss größer 0 sein!Der Schätzwert darf auch nicht 0 sein!');
exit;   // Vorzeitiges Verlassen der Schleife!
end;
Until((Wurzel>0) And(Schaetzwert<>0));

// Heronverfahren nach dem vorliegenden Struktogramm
// eingelesener Schätzwert wird an an übergeben
```

```
an:=Schaetzwert;
//Tabelleneigenschaften einstellen
StringGrid1.DefaultColWidth:=140;
StringGrid1.cells[0,0]:='Nr.:';
StringGrid1.cells[1,0]:='Wert';
// Berechnung nach Heron für 10 Werte
For i:=1 to 10 do
begin
// Gleichung umgesetzt
am:=1/2*(an+wurzel/an);
// Ausgabe in der Tabelle
StringGrid1.cells[0,i]:=IntToStr(i);

StringGrid1.cells[1,i]:=FloatToStr(am);
// neuer Wert für die nächste Berechnung
// nächste Näherung
an:=am;
end;

end;

procedure TF_Haupt.BitBtn1Click(Sender: TObject);
begin
// Button zum Schließen des Programmes
F_Haupt.close;
end;
```

Arbeit mit Meldungen in Delphi!

Einfachste Möglichkeit:

Showmessage('Hier steht der Text');

Besser:

MessageDLG(Meldungstext,Meldungstyp,Knopfauswahl,Hilfsindex):Rückgabewert

Erläuterung:
Meldungstext: Text der im Dialogfenster ausgegeben wird (als String oder Stringvariable)

Meldungstyp:
Vordefinierte Konstanten:
mtWarning (Fensterüberschrift: Warnung)
mtError; (Fensterüberschrift: Fehler)
mtInformation; (Fensterüberschrift:Information)
mtConfirmation; (Fensterüberschrift:Bestätigen)
mtCustum;(Titel ist mit dem Namen der exe- Datei identisch)

zu den ersten Typen werden auch die entsprechenden Windows- Grafiken angezeigt)

Knopfauswahl: (erklärt sich selber, denke ich!)
mbYes
mbNo
mbCancel
mbHelp
mbAbort
mbRetry
mbIgnore
mbAll

mbOkCancel
mbAbortRetryIgnore
mbYesNoCancel

Hilfeindex: Nur von Bedeutung, wenn man eine eigene Hilfe-Datei erstellen möchte. Kann sonst immer 0 sein!

Rückgabe:

mrYes Taste JA mrNo Taste NEIN
mrOK Taste OK
mrCancel Taste ABBRUCH
mrAbort Taste ABBRUCH
mrRetry Taste WIEDERHOLEN
mrIgnore Taste IGNORIEREN
mrAll Taste ALLES wurde gedrückt.

Beispiele:
1.

```
MessageDLG('Hinweistext',mtInformation,[mbOK],0)
```

2.

```
If MessageDlg('Zeile löschen?',mtWarning,[mbYes,mbNo],0)=mrYes then .......
```

3.

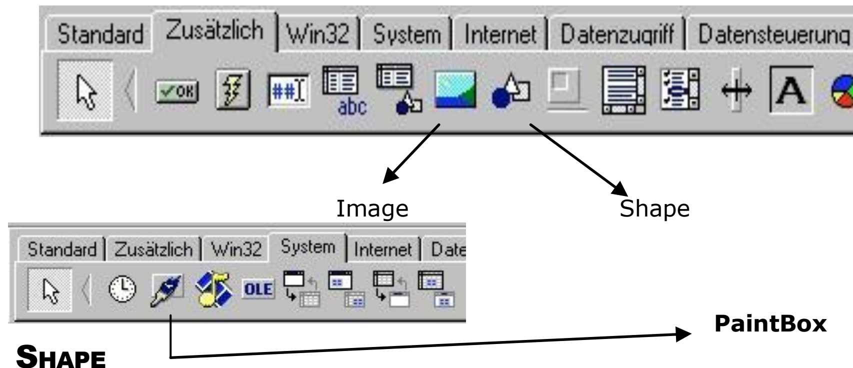

```
case MessageDlg('Knopf drücken!',mtInformation,mbYes NoCancel,0) of
mrYes       :Showmessage('JA- Knopf wurde gedrückt!');
mrNo        :Showmessage('NEIN- Knopf wurde gedrückt!');
mrCancel    :Showmessage('ABBRUCH- Knopf wurde gedrückt!');
end;
```

Einfach mal ausprobieren!

In Delphi gibt es grundsätzlich drei Varianten um Grafik anzuzeigen:

Anzeigen von fertigen Bildern mit Image oder Figuren mit Shape PaintBox Zeichnen von

Einsatz von Grafikmethoden (d. h. Programmierung) auf dem Canvas (engl. Leinwand)



SHAPE

Mit Shape können Sie Formen wie Rechteck, Ellipsen, Kreise auf die gleiche Art wie z. B. ein Button oder ein Label erzeugen.

Im Objektinspektor können Sie die Figur gestalten:

Shape Form auslesen: Rechteck und Quadrat (auch mit abgerundeten Ecken)
Ellipse und Kreis

Brush Füllmuster und Füllfarbe;
Damit diese Optionen zum Vorschein kommen, müssen sie das + vor Brush doppelklicken. Unter Color finden Sie die Farbe, unter Style diverse Füllmuster.

Pen Gestaltung der Randlinie;
+ doppelklicken für die Optionen Color, Style (Strichmuster), Width (Strichbreite in Punkt). Achtung! nur ausgezogene Linien (psSolid) können breiter als 1 gezeichnet werden.

Muster dazu auf den nächsten Seiten!

Auf den folgenden Seiten geht es nun vor allem darum, Grafiken mittels Programmierung herzustellen.

TCanvas dient als Zeichenfläche für Objekte, die sich selbst zeichnen. Die unter Windows üblichen
Steuerelemente wie z.B. Eingabe- oder Listenfelder benötigen keine Zeichenfläche, da sie von Windows gezeichnet werden.
TCanvas stellt Eigenschaften, Ereignisse und Methoden zur Verfügung, die beim Erzeugen von Bildern hilfreich sind, indem sie

die Art des verwendeten Pinsels und Stiftes

sowie die Schriftart festlegen. eine Vielzahl

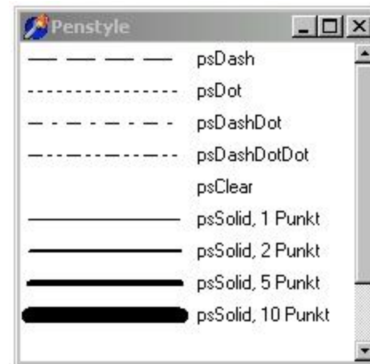
von Fomen und Linien zeichnen und füllen.

Text ausgeben.

Grafiken zeichnen.

eine Reaktion auf Änderungen am aktuellen Bild ermöglichen.

MUSTER ZU PEN



Der Befehl heisst:

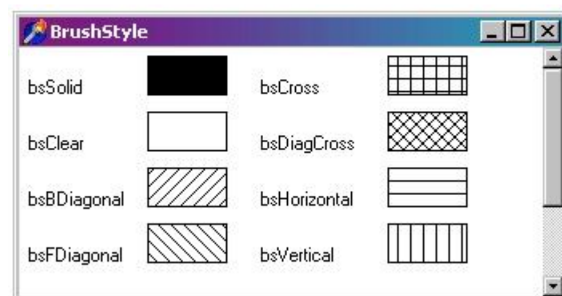
```
Canvas.Pen.Style:=psDash;
```

Der gewählte Style gilt für Linien und Ränder von Figuren; er bleibt eingestellt, bis er durch einen neuen Style-Befehl ersetzt wird.

MUSTER ZU BRUSH

Der Befehl heisst:

```
Canvas.Brush.Style:=psSolid;
```



Die Musterlinien werden in der aktuellen Musterfarbe auf dem bestehenden Hintergrund gezeichnet. Der Rand kann mit Pen.Color oder Pen.Style unabhängig davon gezeichnet werden. Der gewählte Style bleibt eingestellt, bis er durch einen neuen Style-Befehl ersetzt wird.

Farben

Systemfarben (Kastanienbraun)	clAqua	Aqua (Blaugrün)	clMaroon	Maroon
	clBlack	Schwarz	clNavy	Marineblau
	clBlue	Blau	clOlive	Olivgrün
	clDkGray	Dunkelgrün	clPurple	Purpur
	clFuchsia	Fuchsia (rosa)	clRed	Rot
	clGray	Grau	clSilver	Silber
	clGreen	Grün	clTeal	Teal (dunkles Blaugrau)
	clLime	Lime (mittleres Grün)	clWhite	Weiß
	clLtGray	Hellgrün	clYellow	Gelb

Systemfarben werden folgendermassen zugewiesen:

```
Canvas.Pen.Color:=clSilver;
Canvas.Brush.Color:= clBlue;
```

Pen- und Brush-Farben können unabhängig voneinander gewählt werden. Sie bleiben eingestellt, bis sie durch einen neuen Farbbefehl aufgehoben werden.

RGB-Farben Blau

Die Farben werden durch Zahlen dargestellt, die die Anteile an Rot, Grün,

wiedergeben und zwar üblicherweise in Hexadezimalzahlen.

z. B.
```
Canvas.Brush.Color:=$000000FF      (rot)
Canvas.Brush.Color:=$0000FF00      (grün)
Canvas.Brush.Color:=$00FF0000      (blau)
Canvas.Brush.Color:=$00000000      (schwarz)
Canvas.Brush.Color:=$00FFFFFF      (weiss)
```

FF ist die grösste zweistellige Zahl im Hexadezimalsystem, das die 16 Ziffern
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F aufweist und entspricht im Dezimalsystem der Zahl 255.

00	BF	7A	CD
	Blau	Grün	Rot

BF: $11 \supseteq 16 + 15 = 191$ Teile Blau
7A: $7 \supseteq 16 + 10 = 122$ Teile Grün
CD: $12 \supseteq 16 + 13 = 205$ Teile Rot

Ohne Hexadezimalzahlen lässt sich die Farbe $00BF7ACD zuweisen als:

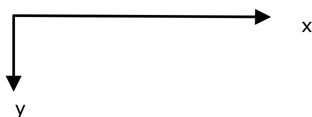
```
Canvas.Pen.Color:=RGB(191,122,205)
```

Mit den Extremwerten:

```
Canvas.Pen.Color:=RGB(0,0,0)            (Schwarz)
Canvas.Pen.Color:=RGB(255,255,255)      (Weiss)
```

KOORDINATEN

Das Koordinatensystem beruht auf der Einheit Punkt oder Pixel. Der Nullpunkt ist in der linken oberen Ecke des Formulars.



Sie sehen im Objektinspektor wie breit (Width) und wie hoch (Hight) das Formular momentan ist.
Falls Sie die Grüsse des Formulars noch verändern, arbeiten Sie besser mit den Zahlen CustomWidth und CustomHight.

(CustomWidth - 20, CustomHight-10) sind die Koordinaten eines Punktes, der vom rechten Rand 20 Pixel und vom untern Rand 10 Pixel Abstand hat.

Der Punkt mit den Koordinaten (CustomWidth div 2 , CustomHight div 2) liegt etwa in der Mitte des Formulars. Beachten Sie das div! Koordinaten müssen ganzzahlig sein!

Tipp: Wenn Sie sich die Koordinaten nicht ganz vorstelllen können:
zeichnen Sie ein Label und lesen Sie die Zahlen im Objektinspektor ab.
(Left , Top) ergibt die Ecke oben links, (Left +Width ,Top +Hight) die Ecke unten rechts.

FIGUREN

Punkt

Auf einen Punkt greifen Sie mit der Eigenschaft Pixels zu:
z. B. den Punkt (50,20) schwarz färben: Canvas.Pixels[50,20]:=clBlack

Da Pixels eine Eigenschaft ist, können Sie damit auch die Farbe eines Punktes ablesen:

```
        var
farbe: Tcolor;
farbe:=Canvas.Pixels[50,20];
```

39

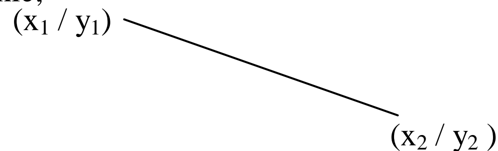
Um abzulesen, wo sich der Stift im Moment befindet, benützen Sie die Eigenschaft PenPos:

```
        var
Ecke: Tpoint;
xwert: LongInt;
Ecke:=Canvas.Pos;
xwert:=Canvas.Pos.X;
```

Geraden

Um Geraden zu zeichnen benötigen Sie zwei Befehle;
Canvas.MoveTo (x1,y1);
Canvas.LineTo (x2,y2);

(x_1 / y_1)



(x_2 / y_2)

Zahlenbeispiel:
Canvas.MoveTo (70, 50);
Canvas.LineTo (410, 120);

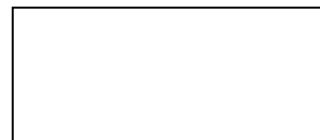
Es ginge auch mit der Methode PolyLine:
Canvas.PolyLine ([Point(x1,y1),Point(x2,y2)]);

Rechteck und Quadrat

Linke obere und rechte untere Grenze angeben:
Canvas.Rectangle (x1,y1,x2,y2);

(x_1 / y_1)

Zahlenbeispiel:
Canvas.Rectangle (70, 50, 170, 100);



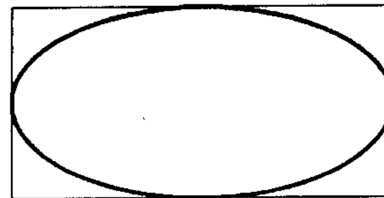
Ein Quadrat hat gleiche Seitenlängen:
Canvas.Rectangle (70, 50, 100, 80);

(x_2 / y_2)

$$\begin{pmatrix} 70 + 30 = 100 \\ 50 + 30 = 80 \end{pmatrix}$$

Weitere Grafikelemente in Delphi

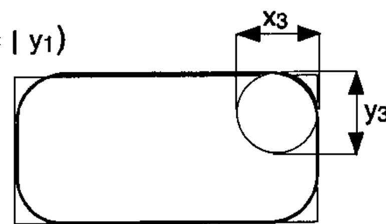
Ellipse und Kreis
Linke obere und rechte untere Grenze des Rechtecks eingeben:
Canvas.Ellipse(x_1,y_1,x_2,y_2);



Zahlenbeispiel:
Canvas.Ellipse(70,50,170,100);

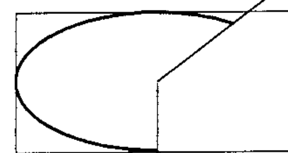
Kreis mit Mittelpunkt (u,v) und Radius r
Canvas.Ellipse(u-r,v-r,u+r,v+r);

Abgerundetes Rechteck
Canvas.RoundRec (x_1,y_1,x_2,y_2,x_3,y_3);



Zahlenbeispiel:
Canvas.RoundRec (70,50,170,100,20,20);

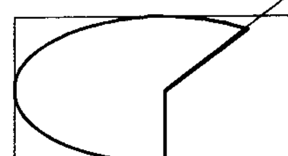
Bogenstück
Canvas.Arc (x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4);



Zahlenbeispiel:
Canvas.Arc (70,50,170,100,150,50,120,100);

Tortenstück

Canvas.Pie (x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4);

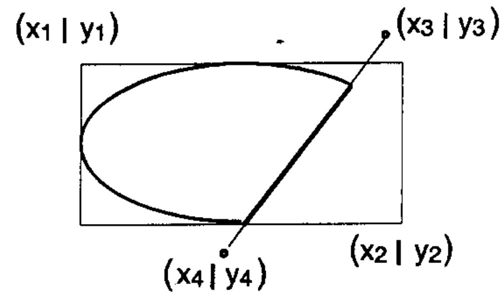


Zahlenbeispiel:
Canvas.Pie (70,50,170,100,150,50,120,100);

Kreissegment

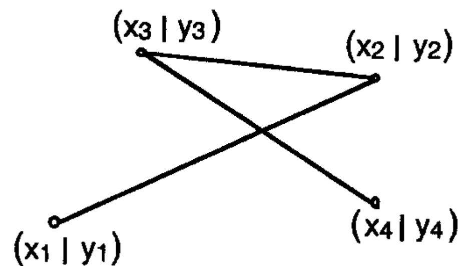
Canvas.Cord (x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4);

Zahlenbeispiel:
Canvas.Cord(70,50,170,100,150,50,120,100);

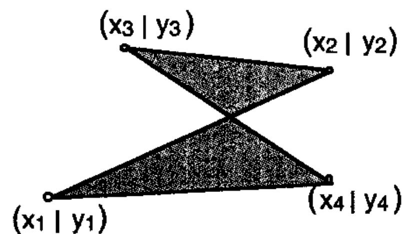


Streckenzug

Der Streckenzug kann nicht gefüllt werden!
Canvas.PolyLine([Point (x_1,y_1),Point(x_2,y_2),…]);



Vieleck
Ein Polygon schliesst sich selbst und kann gefüllt werden!
Canvas.Polygon ([Point (x_1,y_1),Point(x_2,y_2),…]);



Textausgabe
Einfachste Methode um Text auf den Bildschirm zu bringen.
Als Schriftart wird der aktuelle Wert voon Font verwendet. Die Bezeichnung der Lage geben die Koordinaten der Ecke oben links an.
Beispiel:
Canvas.Font.Height:=20;
Canvas.TextOut(20,10,'Hallo Welt!');

Damit das nicht (besonders bei vielen Elementen) in enorme Tipparbeit ausartet, gibt es dafür auch eine Abkürzung:

```
with canvas do begin
Canvas.Font.Height:=20;
Canvas.TextOut(20,10,'Hallo Welt!');
Canvas.Cord(70,50,170,100,150,50,120,100);
…
…
end;
```

Verzweigungen

if-else

Es gibt kaum ein Programm, bei dem immer alle Befehle hintereinander ausgeführt werden. Verzweigungen sind ein häufig eingesetztes Mittel. Es handelt sich hierbei um Fallunterscheidungen, die in der Regel mit if durchgeführt werden:

```
if  then ;
```

oder

```
if  then  else ;
```

Eine Anweisung kann dabei wiederum aus einer neuen if-Bedingung bestehen.

Beispiel:

```
if x>0 then ...
else if x<0 then ...
else ...;
```

Das Beispiel bedeutet Folgendes: Ist x größer als Null, wird das ausgeführt, was hinter dem ersten `then` steht (die drei Punkte). Handelt es sich dabei um mehr als einen Befehl, muss der Block mit `begin` und `end` umgeben werden.

`else`, zu Deutsch "sonst", leitet eine Alternative ein. Wenn also die erste Bedingung nicht erfüllt ist, wird die zweite geprüft, hier, ob x vielleicht kleiner als Null ist. Trifft auch diese Bedingung nicht zu, bleibt noch ein else ohne Bedingung. Die letzten drei Punkte werden also immer dann ausgeführt, wenn die ersten beiden Bedingungen nicht zutreffen.

Wäre bereits die erste Bedingung erfüllt gewesen, so wären die folgenden `else`-Abschnitte gar nicht mehr geprüft worden.

Selbstverständlich muss so ein `if`-Block keine "`else if`"- oder "`else`"-Alternativen bieten. Das kommt immer auf die Situation an. Da sich das Ganze aber sehr stark an mathematische Logik anlehnt, dürfte die Notation nicht allzu schwer fallen.

Was allerdings zu beachten ist: In Delphi steht hinter dem letzten Befehl vor dem Wörtchen else kein Strichpunkt (Semikolon) wie sonst üblich - im Gegensatz zu C++.

Noch ein Beispiel, wobei jeweils mehrere Befehle ausgeführt werden:

```
var eingabe: integer;
...
if eingabe=1 then begin
  eingabe := 0;
  ausgabe := 'Sie haben eine 1 eingegeben';
end //kein Strichpunkt!
else if eingabe=2 then begin
  eingabe := 0;
  ausgabe := 'Sie haben eine 2 eingegeben';
end
else begin
  eingabe := 0;
  ausgabe := 'Sie haben eine andere Zahl als 1 oder 2 eingegeben';
end;
```

Hier sieht man besonders den Sinn von else. Wären die drei Fallunterscheidungen durch drei getrennte `if`-Abfragen dargestellt worden, dann hätten wir folgendes Problem: Angenommen `eingabe` ist 1, so ist die erste Bedingung erfüllt. Da hier `eingabe` jedoch auf 0 gesetzt wird, träfe nun auch die dritte Bedingung zu. Im obigen Beispiel mit else stellt das kein Problem dar.

case-Verzweigung

Müssten wir in obigem Beispiel mehr als nur zwei Zahlen prüfen, hätten wir ganz schön Tipparbeit. Für solche abzählbaren (ordinalen) Typen wie Integer und Char gibt es in Delphi eine Abkürzung:

```
case eingabe of
  1: ausgabe := 'Sie haben 1 eingegeben';
  2: ausgabe := 'Sie haben 2 eingegeben';
  3: ausgabe := 'Sie haben 3 eingegeben';
  else ausgabe := 'Sie haben nicht 1, 2 oder 3 eingegeben';
end;
```

Zugegeben, das Beispiel ist nicht besonders sinnvoll, da die Variable `eingabe` direkt in einen String umgewandelt werden könnte. Allerdings stellt es gut die Funktionsweise von `case` dar. Zu beachten ist, dass am Ende eines `Case`-Blocks ein `end` stehen muss. Gehören mehrere Anweisungen zusammen, können sie wie bei `if` durch `begin` und `end` als zusammengehörig gekennzeichnet werden.

Bei `case` steht (im Gegensatz zu if) vor dem Schlüsselwort else ein Strichpunkt!

Mit `case` ist auch Folgendes möglich:

```
case eingabe of
  1,3,5,7,9: ausgabe := 'Sie haben eine ungerade Zahl kleiner als 10 
eingegeben';
  2,4,6,8,0: ausgabe := 'Sie haben eine gerade Zahl kleiner als 10 
eingegeben';
  10..20:    ausgabe := 'Sie haben eine Zahl zwischen 10 und 20 
eingegeben';
end;
```


So etwas ist natürlich auch bei `if`-Bedingungen möglich, dazu allerdings noch einen Einschub zum Thema Notation von logischen Bedingungen.

Komplexe Datentypen

Zum Pascal-Sprachumfang gehören nicht nur einfache Datentypen, wie sie im Abschnitt Variablen und Konstanten beschrieben sind, sondern auch zusammengesetzte.

Mengentypen

Um in einer einzigen Variablen eine unterschiedliche Menge an Werten des gleichen Typs zu speichern, gibt es Mengentypen. Es ist eine Menge an möglichen Werten vorgegeben, aus der eine beliebige Anzahl (keiner bis alle) in der Variablen abgelegt werden kann. Folgendermaßen wird eine solche Mengenvariable deklariert:


```
var zahlen: set of 1..10;
```

Damit können der Variablen zahlen Werte aus der Menge der Zahlen von 1 bis 10 zugewiesen werden - mehrere gleichzeitig oder auch gar keiner:

```
zahlen := [5, 9];            // zahlen enthält die Zahlen 5 und 9
zahlen := [];                // zahlen enthält überhaupt keine Werte
zahlen := [1..3];            // zahlen enthält die Zahlen von 1 bis 3
zahlen := zahlen + [5];      // zahlen enthält die Zahlen 1, 2, 3, 5
zahlen := zahlen - [3..10]; // die Zahlen von 3 bis 10 werden aus der
Menge
                             // entfernt, es bleiben 1 und 2
```

Um nun zu prüfen, ob ein bestimmter Wert in der Menge enthalten ist, wird der Operator in verwendet:

```
if 7 in zahlen then ...
```

In einem Set sind nur Werte mit der Ordnungsposition von 0 bis 255 möglich.

Arrays

Müssen mehrere Werte des gleichen Typs gespeichert werden, ist ein Array (zu deutsch Feld oder Liste) eine praktische Lösung. Ein Array hat einen Namen wie eine Variable, jedoch gefolgt von einem Index in eckigen Klammern. Über diesen Index kann man auf die einzelnen Werte zugreifen. Am einfachsten lässt sich das mit einer Straße vergleichen, in der sich lauter gleiche Häuser befinden, die sich jedoch durch ihre Hausnummer (den Index) unterscheiden.

Eine Deklaration der Art

```
var testwert: array [0..10] of integer;
```

bewirkt also, dass wir quasi elf verschiedene Integer-Variablen bekommen. Man kann auf sie über den Indexwert zugreifen:

```
testwert[0] := 15;
testwert[1] := 234;
```

usw.

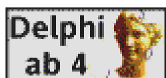
Vorteil dieses Indexes ist, dass man ihn durch eine weitere Variable ersetzen kann, die dann in einer Schleife hochgezählt wird. Folgendes Beispiel belegt alle Elemente mit dem Wert 1:


```
for i := 0 to 10 do
  testwert[i] := 1;
```

Auf Schleifen wird jedoch in einem gesonderten Kapitel eingegangen.

Bei dem vorgestellten Array handelt es sich genauer gesagt um ein eindimensionales, statisches Array. "Eindimensional", weil die Elemente über nur einen Index identifiziert werden, und "statisch", weil Speicher für alle Elemente reserviert wird. Legt man also ein Array für Indexwerte von 1 bis 10000 an, so wird für 10000 Werte Speicher reserviert, auch wenn während des Programmablaufs nur auf zwei Elemente zugegriffen wird. Außerdem kann die Array-Größe zur Programmlaufzeit nicht verändert werden.

Dynamische Arrays



Wenn schon so viel Wert auf die Eigenschaft statisch gelegt wird, muss es ja eigentlich auch etwas Dynamisches geben. Und das gibt es auch, zumindest seit Delphi 4: die dynamischen Arrays.

Der erste Unterschied findet sich in der Deklaration: Es werden keine Grenzen angegeben.

```
var dynArray: array of integer;
```

dynArray ist nun prinzipiell eine Liste von Integer-Werten, die bei Index Null beginnt.

Zum Hintergrundverständnis: Während statische Arrays direkt die einzelnen Werte beinhalten, enthält ein dynamisches Array nur Zeiger auf einen Arbeitsspeicherbereich. In der Anwendung ist das nicht zu merken, es ist keine spezielle Zeigerschreibweise nötig. Nur an einer Stelle bemerkt man das interne Vorgehen: Bevor man Werte in das Array stecken kann, muss man Speicher für die Elemente reservieren. Dabei gibt man an, wie groß das Array sein soll:

```
SetLength(dynArray, 5);
```

Nun kann das Array fünf Elemente (hier Integer-Zahlen) aufnehmen.

Man beachte: Da die Zählung bei Null beginnt, befindet sich das fünfte Element bei Indexposition 4!

Der Zugriff erfolgt ganz normal:

```
dynArray[0] := 321;
```

Damit es mit der Unter- und vor allem der Obergrenze, die ja jederzeit verändert werden kann, keine Probleme gibt (Zugriffe auf nicht (mehr) reservierten Speicher), lassen sich Schleifen am einfachsten so realisieren:


```
for i := 0 to high(dynArray) do
  dynArray[i] := 0;
```

Dadurch werden alle Elemente auf 0 gesetzt. `high(dynArray)` entspricht dem höchstmöglichen Index. Über `length(a)` lässt sich die Länge des Arrays ermitteln, welche immer `high(dynArray)+1` ist. Dabei handelt es sich um den Wert, den man mit `SetLength` gesetzt hat.

Da die Länge des Arrays jederzeit verändert werden kann, könnten wir sie jetzt mit

```
SetLength(dynArray, 2);
```

auf zwei verkleinern. Die drei hinteren Werte fallen dadurch weg. Würden wir das Array dagegen vergrößern, würden sich am Ende Elemente mit undefiniertem Wert befinden.

Mehrdimensionale Arrays

Wenn es eindimensionale Arrays gibt, muss es auch mehrdimensionale geben. Am häufigsten sind hier wohl die zweidimensionalen. Man kann mit ihnen z. B. ein Koordinatensystem oder Schachbrett abbilden. Die Deklaration ist wie folgt:

```
var koordinate: array [1..10, 1..10] of integer;
```

Es werden also 10x10=100 Elemente angelegt, die jeweils einen Integer-Wert aufnehmen können. Für den Zugriff auf einzelne Werte sind zwei Schreibweisen möglich:

```
  koordinate[1,6] := 34;
  koordinate[7][3] := 42;
```

Records

Records entsprechen von der Struktur her einem Datensatz einer Datenbank - nur dass sie, wie alle bisher aufgeführten Variablen, nur zur Laufzeit vorhanden sind. Wenn wir unterschiedliche Daten haben, die logisch zusammengehören, können wir sie sinnvollerweise zu einem Record zusammenfassen. Beispiel: Adressen.

```
var Adresse: record
  name: string;
  plz: integer;
  ort: string;
end;
```

Die Variable Adresse wird also in drei "Untervariablen" aufgegliedert.

Folgendermaßen greift man auf die einzelnen Felder zu:


```
adresse.name := 'Hans Müller';
adresse.plz := 12345;
adresse.ort := 'Irgendwo';
```

Damit das nicht (besonders bei vielen Elementen) in enorme Tipparbeit ausartet, gibt es dafür auch eine Abkürzung:

```
with adresse do begin
  name := 'Hans Müller';
  plz := 12345;
  ort := 'Irgendwo';
end;
```

Prozeduren und Funktionen

Komplexe Probleme lassen sich übersichtlich in kleinere Teilprobleme aufspalten. Dazu eignen sich Funktionen und Prozeduren. Eine Prozedur oder Funktion wird über ihren Namen angesprochen. Prozeduren/Funktionen sind Zusammenfassungen von Anweisungen, die im Hauptprogramm oder einer anderen Prozedur/Funktion möglicherweise oft gleichartig vorkommen, an die Werte übergeben werden können und die Werte weitergeben können.

Eine Prozedurdefinition beginnt mit der Prozedurdeklaration. Sie beginnt mit dem Schlüsselwort "**procedure**". Dahinter folgt der Name der Prozedur, evtl. gefolgt von der Angabe der formalen Parameter in runden Klammern.
Die formalen Parameter (mit ihrem Datentyp hinter einem Doppelpunkt) sind durch Komma getrennt. Es gibt zwei Arten. Steht ein "**var**" vor dem Parameter, dann handelt es sich um einen Referenzparameter, sonst ist es ein Wertparameter. Wertparameter verändern nichts im Hauptprogramm. Referenzparameter nutzen (und verändern) gleichnamige Variablen im Hauptprogramm.

Hier ein ganz einfaches Beispiel:

```
implementation

{$R *.DFM}

procedure Ruf;
begin
  ShowMessage('Es klappt!');
end;

procedure TForm1.Button1Click(Sender: TObject);

begin
  Ruf;
end;

end.
```

Funktionen können ein (oder mehrere) Argument(e) haben und geben einen einzigen Wert weiter. Eine Funktionsdefinition beginnt mit der Funktionsdeklaration. Sie beginnt mit dem Schlüsselwort "**function**". Dahinter folgt der Name der Funktion, gefolgt von der Angabe der formalen Parameter in runden Klammern.
Die formalen Parameter (mit ihrem Datentyp hinter einem Doppelpunkt) sind durch Komma getrennt. Es gibt zwei Arten. Steht ein "**var**" vor dem Parameter, dann handelt es sich um einen Referenzparameter, sonst ist es ein Wertparameter. Wertparameter verändern nichts im Hauptprogramm. Referenzparameter nutzen (und verändern) gleichnamige Variablen im Hauptprogramm.
Hinter der runden Klammer steht nach einem Doppelpunkt der Datentyp des Wertes, den die Funktion liefern soll.

```
implementation

{$R *.DFM}

function Doppelt(a: Integer): Integer;

begin
  Doppelt := a * 2;
end;
```

Nach der Funktionsdeklaration können hinter "**var**" Variablen deklariert werden.
Die Arbeitsweise der Funktion wird im Funktionsrumpf festgelegt, der zwischen "**begin**" und "**end**" eingeschlossen wird.

Hier ein Beispiel für eine etwas umfangreichere Funktion (Fakultät von n):

```
implementation

{$R *.DFM}

function Fak(n: Integer): LongInt;

var i: Integer;

begin
  Fak := 1;
  for i := 1 to n do Fak := Fak * i;
end;
```

Hier ein Beispiel für eine etwas umfangreichere Prozedur
(Zwei Integer-Variable a und b werden mit `Tausch(a,b);` eingegeben und mit vertauschten Inhalten wieder ausgegeben.):

```
implementation

{$R *.DFM}

procedure Tausch(var x,y: Integer);

var hilf: Integer;

begin
  hilf := x;
  x := y;
  y := hilf;
end;
```

Besonderheit von Funktionen gegenüber Prozeduren:

- Der Typ des Resultatwertes wird bei Funktionen hinter der Parameterliste vereinbart.
- Im Funktionskörper muss einer Variablen mit den Namen der Funktion ein Wert des entsprechenden Typs zugewiesen werden.
- Der Funktionsaufruf erfolgt innerhalb eines Ausdrucks, während eine Prozedur alleine eine Anweisung sein kann.
- Eine Funktion benutzt man immer dann, wenn **ein** Wert zu berechnen ist. Das ist guter Programmierstil(, denn eigentlich könnte man alles mit Prozeduren machen).

Prozeduren sollten aber der Form wegen immer **deklariert** und dann **implementiert** werden!

Deklaration von Funktionen und Prozeduren

```
unit U_Haupt;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  ExtCtrls, StdCtrls;
type
  TF_Anwendung = class(TForm)
    P_Ueberschrift: TPanel;
    E_Laenge: TEdit;
    E_Breite: TEdit;
    E_Umfang: TEdit;
    E_Flaeche: TEdit;
    E_Diagonale: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    Label6: TLabel;
    Label7: TLabel;
    Label8: TLabel;
    Label9: TLabel;
    Label10: TLabel;
    B_Berechnen: TButton;
    B_Beenden: TButton;
    procedure B_BeendenClick(Sender: TObject);
  private
    { Private-Deklarationen}
    {Anmeldung aller privaten Funktionen und Prozeduren!!!!!}

    function FlaecheRechteck(Seite_a,Seite_b:Single):Single ;
  public
    { Public-Deklarationen}
  end;
var
  F_Anwendung: TF_Anwendung;
 {Hier werden alle globalen Variablen für das Programm angemeldet!}
implementation
{Alle Prozeduren und Funktionen werden hier programmiert!!!!  }
{$R *.DFM}
{Beachte:
Wenn eine Funktion oder Prozedur deklariert wurde, muss bei der
Implementation die Typenklasse mit angegeben werden
Hier TF_Anwendung }
function FTF_Anwendung.FlaecheRechteck(Seite_a,Seite_b:Single):Single
 begin
 Result:=Seite_a*Seite_b;
 { Es wäre auch folgendes möglich:
 FlaecheRechteck:= Seite_a*Seite_b;    }
```

51

Routinen für die Zeichenketten-Bearbeitung

Funktion	Bedeutung	Beispiel / Bemerkung
CONCAT	Verbindet mehrere Strings	s := 'Dies ist ein Test'; CONCAT('Achtung: ',s); ergibt: 'Achtung: Dies ist ein Test' (gleiche Wirkung wie: s := 'Achtung: ' + s;)
COPY	Kopiert einen Teil aus einem String	s := 'Dies ist ein Test'; COPY(s, 14, 4); ergibt: 'Test'
DELETE	Löscht einen Teil innerhalb eines String	s := 'Dies ist ein Test'; DELETE(s, 10, 4); ergibt: 'Dies ist Test'
INSERT	Fügt eine String in einen anderen ein	s := 'Dies ist ein Test'; INSERT('einfacher ',s, 14); ergibt: 'Dies ist ein einfacher Test'
LENGTH	Ermittelt die Anzahl der Zeichen in einem String	s := 'Test'; t:= ''; LENGTH(s) ergibt: 4 LENGTH(t) ergibt: 0
POS	Sucht einen String in einem anderen und gibt die Position des ersten Zeichens der Fundstelle zurück	s := 'Dies ist ein Test'; POS('T', s); ergibt: 14 POS('t', s); ergibt: 8
STR	Wandelt eine Zahl in einen String (Folge von Zeichen) um	i := 3; r := 0.3; STR(i, s) ergibt: s ist '3' STR(I:3, s) ergibt: s ist ' 3' STR(r, s) ergibt: s ist '3.0000' STR(r:3, s) ergibt: s ist '3.0E-0001' (besser ist IntToStr)
LOWERCASE	Konvertiert alle Zeichen in einem String in Kleinbuchstaben	s := 'Dies ist ein Test 1234'; LOWERCASE(s); ergibt: 'dies ist ein test 1234'
UPPERCASE	Konvertiert alle Zeichen in einem String in Großbuchstaben	s := 'Dies ist ein Test 1234'; UPPERCASE(s); ergibt: 'DIES IST EIN TEST 1234'
TRIM	Entfernt Leerzeichen in einem String, die am Anfang oder Ende stehen	
TRIMLEFT	Entfernt alle Leerzeichen am Anfang des Strings	
TRIMRIGHT	Entfernt alle Leerzeichen am	

	Ende des Strings	
VAL	Wandelt einen String in eine Zahl um, falls möglich	(besser ist StrToFloat)

Darstellung von Gleitkommazahlen

Rechenergebnisse, also Zahlen, will man ja meist irgendwie anzeigen lassen, damit man sie lesen kann. Dazu muss man sie in eine Zeichenkombination, also einen String, umwandeln. Die einfachste Möglichkeit ist die Funktion **FloatToStr(Zahl)** .

Allerdings wird so die Zahl mit ihrer ganzen Genauigkeit mit möglicherweise 15 Stellen angezeigt, auch wenn das gar keinen Sinn macht.

Mit der Funktion FloatToStrF(Zahl,Formattyp,Genauigkeit,Digits) kann man selbst bestimmen, wie die Zahl dargestellt werden soll.

Darstellung von Gleitkommazahlen

Dargestellt wird die Zahl : 1000*Wurzel(2)

FloatToStr(Zahl)	1414,21362304688
FloatToStrF(Zahl,ffGeneral,8,0)	1414,2136
FloatToStrF(Zahl,ffGeneral,4,0)	1414
FloatToStrF(Zahl,ffExponent,8,3)	1,4142136E+003
FloatToStrF(Zahl,ffExponent,4,2)	1,414E+03
FloatToStrF(Zahl,ffFixed,8,3)	1414,214
FloatToStrF(Zahl,ffFixed,4,2)	1414,00
FloatToStrF(Zahl,ffNumber,8,3)	1.414,214
FloatToStrF(Zahl,ffNumber,4,2)	1.414,21
FloatToStrF(Zahl,ffCurrency,8,2)	1.414,21 €
FloatToStrF(Zahl,ffCurrency,4,2)	1.414,00 €

Löschen Schließen

Beispiele für die Behandlung von Datum und Uhrzeit

Delphi verwaltet das Datum als Anzahl der Tage seit dem 1. Januar 001. Im Gleitkommateil dieser Zahl steckt die Uhrzeit.
Anzeigen lassen kann man sich Datum und Uhrzeit ganz einfach, wenn man die Timer-Komponente in die Form zieht und folgenden Code schreibt:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  Label1.Caption := DateTimeToStr(Now);
end;
```


Darüber hinaus bietet Delphi einige Routinen zur Arbeit mit Datum und Uhrzeit:

Funktion	***Bedeutung***
`Date`	Liefert das aktuelle Datum im TDateTime-Format zurück
`DateTimeToStr`	Wandelt das TDateTime-Format in einen String um
`DateTimeToString`	Wandelt das TDateTime-Format in einen String um, berücksichtigt dabei aber die Formatierungsinformationen, die als Argument übergeben wurden
`DateToStr`	Wandelt einen Datumsformat-Wert in einen String um
`DayOfWeek`	Liefert den aktuellen Wochentag zurück
`DecodeDate`	Decodiert das angegebene Datum
`EncodeDate`	Wandelt die einzeln übergebenen Werte in das Datums-/Zeit-Format um
`DecodeTime`	Decodiert die angegebene Uhrzeit
`FormateDateTime`	Wandelt ein TDateTime-Format in einen String um, berücksichtigt dabei die Formatierungsinformationen
`Now`	Liefert das aktuelle Datum und die Uhrzeit
`StrToDate`	Wandelt eine Zeichenkette in das Datumsformat um
`StrToDateTime`	Wandelt eine Zeichenkette in das TDateTime-Format um
`StrToTime`	Wandelt eine Zeichenkette in das Zeitformat um
`Time`	Liefert die aktuelle Uhrzeit zurück
`TimeToStr`	Wandelt das Zeitformat in eine Zeichenkette um

Beispiel für die Zeitmessung unter Verwendung der Funktion time

```
…
Var Anfang, Ende: Single;
…
Anfang:=Time,
…( Hier wird das Programm ausgeführt)
Ende:=Time;
E_Zeit.Text:=FloatToStrF((Ende-Anfang)*24*3600,ffFixed,10,2);
```

Arithmetische Funktionen für reelle Zahlen

Mit den folgenden arithmetischen Routinen lassen sich die unterschiedlichsten Integer- und Gleitkommaberechnungen und andere arithmetische oder erweiterte mathematischen Operatoren durchführen:

Funktion	Bedeutung
ABS	Absolutwert des Arguments
ARCTAN	Arcustangens des Arguments
COS	Kosinus des Arguments, das in Bogenmaß übergeben wurde
EXP	Exponentialwert des Arguments
FRAC	Nicht-ganzzahliger Anteil des Arguments
INT	Ganzzahliger Anteil des Arguments
LN	Natürlicher Logarithmus des Arguments
PI	Wert der Konstanten Pi
RANDOM	Liefert eine Pseudozufallszahl
RANDOMIZE	Initialisiert den Zufallsgenerator
ROUND	Das Real-Argument wird auf einen ganzzahligen Wert gerundet
SIN	Sinus des Arguments, das in Bogenmaß übergeben wurde
SQR	Quadrat des Arguments
SQRT	Quadratwurzel des Argumentss
TRUNC	Wandelt eine Realwert in einen Integer-Wert um, indem die Nachkommastellen verworfen werden

Beispiele für das Rechnen mit ganzen Zahlen

Viele Rechenoperationen machen nur Sinn, wenn sie im Bereich der ganzen Zahlen ausgeführt werden. Dazu gehört z. B. die Berechnung von Koordinaten eines grafischen Objekts.

Es gibt zwei Möglichkeiten, aus reellen Zahlen ganze Zahlen zu machen:

```
IntegerZahl := ROUND(reelleZahl);
```

rundet die reelle Zahl auf einen ganzzahligen Wert.

```
IntegerZahl := TRUNC(reelleZahl);
```

wandelt die reelle Zahl in einen ganzzahligen Wert, indem die Nachkommastellen verworfen werden.

Für die Division ganzer Zahlen gibt es zwei Operationen:

```
IntegerZahlc := IntegerZahla DIV IntegerZahlb;
```

Der DIV-Operator lässt die Nachkommastellen unter den Tisch fallen (wie TRUNC). Gebraucht wird diese Operation z. B., wenn man berechnen will, wieviele ganze Stunden in 10000 Sekunden sind (10000 DIV 3600), nämlich 2 Stunden.

```
IntegerZahlc := IntegerZahla MOD IntegerZahlb;
```

Der MOD-Operator liefert den Rest einer Division. Gebraucht wird diese Operation z. B., wenn man berechnen will, wie viele Sekunden in 1000 Sekunden mehr sind als volle Stunden (1000 DIV 3600), nämlich 2800 Sekunden, um daraus vielleicht noch die vollen Minuten abtrennen zu können (2800 DIV 60), also 46 Minuten. dann bleiben noch 2800 MOD 60 (gleich 40) Sekunden.

Beispiele für die Behandlung von aufzählbaren Typen

Der Typ der Laufvariablen z. B. einer FOR-Schleife kann außer Integer auch Char sein. Ebenso können an anderen Stellen noch weitere Aufzählungstypen wie z. B. Wochentage oder Monatsnamen benutzt werden. Für all diese Ordinalwerte gibt es bei Object-Pascal vielfältige Funktionen.

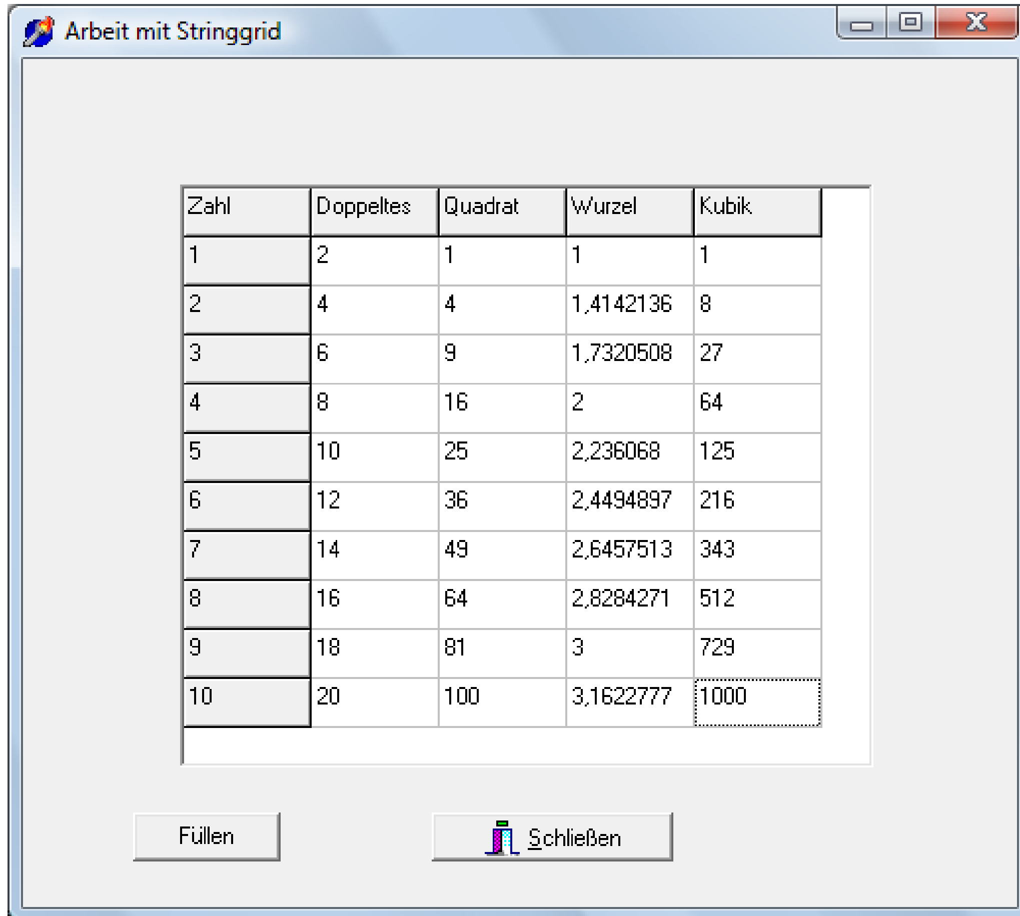
Funktion	***Bedeutung***
DEC	Vermindert eine Variable um einen bestimmten Wert
INC	Erhöht eine Variable um einen bestimmten Wert
ODD	Prüft, ob das Argument eine ungerade Zahl ist
PRED	Liefert den Vorgänger des Arguments
SUCC	Liefert den Nachfolger des Arguments
CHR	Wandelt eine Ordinalzahl in das dazugehörige Zeichen um
HIGH	Ergibt den höchsten Wert im Bereich des Arguments
LOW	Ergibt den niedrigsten Wert im Bereich des Arguments
ORD	Wandelt einen ordinalen Typ in die dazugehörige Ordinalzahl um

Beispiel	***Ergebnis***
x := 'c'; DEC(x)	x ist 'b'
x := 'A'; INC(x)	x ist 'B'
x := 10; DEC(x,4)	x ist 6
x := 'A'; INC(x,3)	x ist 'D'

ODD(x)	x := StrToInt(Edit1.Text);
	if ODD(x) then
	Label1.Caption := 'ungerade!'
	else
	Label1.Caption := 'gerade!';
	end;

PRED('C')	'B'
SUCC(5)	6
CHR(65)	'A'
ORD('A')	65

Arbeit mit Stringgrid (Ausgabe in Tabellen)



Zahl	Doppeltes	Quadrat	Wurzel	Kubik
1	2	1	1	1
2	4	4	1,4142136	8
3	6	9	1,7320508	27
4	8	16	2	64
5	10	25	2,236068	125
6	12	36	2,4494897	216
7	14	49	2,6457513	343
8	16	64	2,8284271	512
9	18	81	3	729
10	20	100	3,1622777	1000

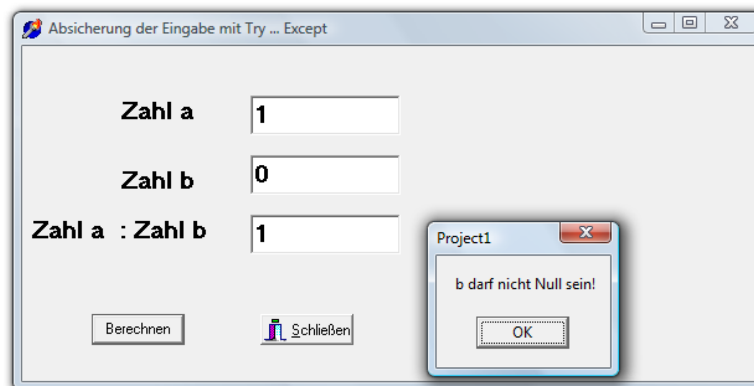
Quelltext:

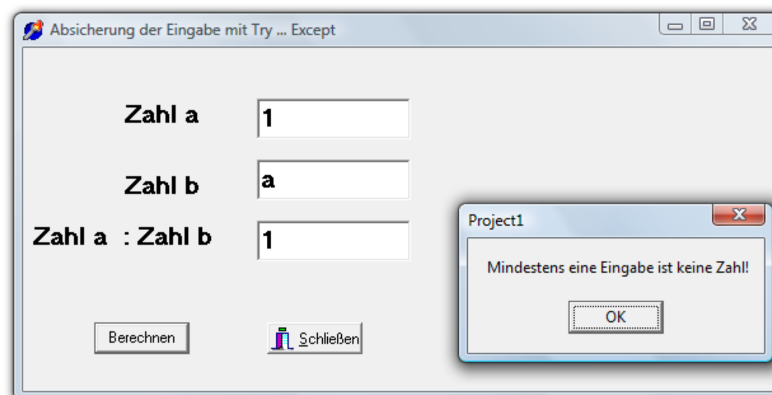
```
procedure TForm1.B_FuellenClick(Sender: TObject);
var i, j: Integer;
begin
    StringGrid1.Cells[0,0] := 'Zahl';
    StringGrid1.Cells[1,0] := 'Doppeltes';
    StringGrid1.Cells[2,0] := 'Quadrat';
    StringGrid1.Cells[3,0] := 'Wurzel';
    StringGrid1.Cells[4,0] := 'Kubik';
    StringGrid1.Cells[5,0] := 'Kehrwert';

    for i := 1 to 10 do
    begin
    StringGrid1.Cells[0,i] := IntToStr(i);
    StringGrid1.Cells[1,i] := IntToStr(2*i);
    StringGrid1.Cells[2,i] := IntToStr(i*i);
    StringGrid1.Cells[3,i] := FloatToStrF(sqrt(i),ffGeneral,8,0);
    StringGrid1.Cells[4,i] := IntToStr(i*i*i);
    StringGrid1.Cells[5,i] := FloatToStrF(1/i,ffGeneral,8,0);
    If i<10 then Stringgrid1.RowCount:= Stringgrid1.RowCount+1;
    end;
end;
```

Eingabekontrolle

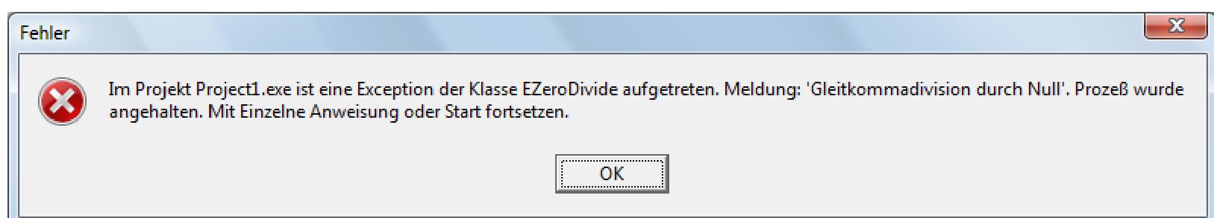
Es ist ein Projekt zu entwickeln, bei dem Fehlermeldungen abgefangen werden, wenn statt einer Zahl eine falsche Zeichenkombination eingegeben wurden oder wenn eine Division durch Null verursacht würde.

Es soll dann eine Meldung erscheinen und alle Werte wieder auf '1' zurück gesetzt werden.

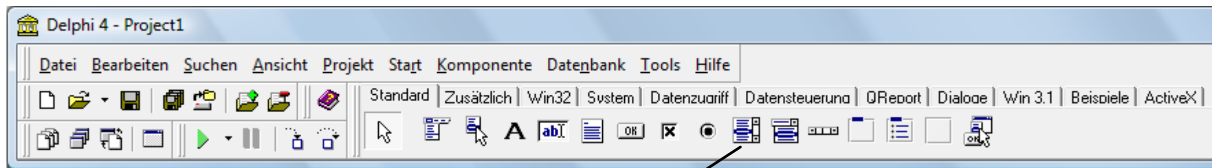




<u>Achtung:</u> Die Auswirkungen von try except werden erst in dem Programm wirksam. Im Probelauf von Delphi erfolgt weiterhin ein die Fehlermeldung



Listbox- Komponente



ListBox

61

Mit der Eigenschaft ***ItemIndex*** der Komponente ***ListBox*** kann festgestellt werden, ob ein Eintragung mit vom Benutzer ausgewählt wurde. Der Funktionswert liefert die Nummer des gewählten Elements, wobei die Zählung mit 0 beginnt. Ist das Ergebnis -1, wurde kein Eintrag ausgewählt. Immer, wenn das **OnMouseUp**-Ereignis der ***ListBox***-Komponente eintritt, wird die Abfrage durchgeführt, um die ***Enabled***-Eigenschaft von "ButtonWeg" zu aktualisieren.

Zur Laufzeit kann ***ItemIndex*** auch ein Wert zugewiesen werden. Dies bewirkt eine Selektion der Eintragung mit diesem Index.

Jede ***ListBox*** hat eine Eigenschaft mit dem Namen ***Items*** vom Typ *Tstrings*. Die wichtigsten Methoden zeigt die folgende Tabelle (F = Function; P = Procedure).

Methode	F/P	Beschreibung der Methode
Add (const S: string): Integer;	F	Einfügen eines Strings in die Stringliste Funktionswert: Position des Strings
Insert (Index: Integer; const S: string);	P	Fügt den String an Position *Index* ein
Clear;	P	Löscht alle Zeilen in Listboxen
Delete (Index: Integer);	P	Löscht den String an Position *Index*
Count : Integer;	F	Ergibt die Anzahl der Einträge

Der Index einer Stringliste beginnt mit 0, d.h. die erste Zeile in ***Listboxen*** hat den Indexwert 0.

Beispielprogramm

```
procedure TFormTextList.ListBoxTextMouseUp(Sender: Tobject;
             Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  ButtonWeg.enabled := ListboxText.itemIndex >= 0;
end;

procedure TFormTextList.ButtonDazuClick(Sender: TObject);
begin
  ListBoxText.items.Add(EditEingabe.Text);
  ButtonLeer.enabled := TRUE;
end;

procedure TFormTextList.ButtonWegClick(Sender: TObject);
begin
  ListBoxText.items.Delete(ListBoxText.ItemIndex);
  ButtonWeg.enabled:= FALSE;
  ButtonLeer.enabled := ListBoxText.items.count >= 1;
end;

procedure TFormTextList.ButtonLeerClick(Sender: TObject);
begin
  ListBoxText.Clear;
  ButtonLeer.enabled := FALSE;
  ButtonWeg.enabled := FALSE;
end;

procedure TFormListe.RadioButtonSortClick(Sender: TObject);
begin
  ListBoxText.sorted:= TRUE;
end;

procedure TFormListe.RadioButtonAnsEndeClick(Sender: TObject);
begin
  ListBoxText.sorted:= FALSE;
end;
```

Delphi- Befehle

Es ist unmöglich alle Befehle als Referenz darzustellen. Die Befehlsreferenz in Delphi ist die Onlinehilfe. Hilfe zu den Delphi- Befehlen kannst du folgendermaßen sehr einfacherhalten:

- Schreibe den Befehl in den Editor
- Stelle den Schreibcursor auf den entsprechenden Befehl
- Drücke bei gehaltener STRG- bzw. CTRL- Taste die F1 Taste