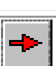
Delphi-Programmierkurs M. Pabst, Lessing-Gymnasium Plauen

Inhaltsverzeichnis



HTML-Version: 25.02.2001 Download: delphtml.zip (25.02.2001)

original URL: http://www.plauener.de/delphi

0. Vorbemerkungen

1. Einführung in Delphi

- 1.1. Was ist Delphi?
- 1.2. Algorithmen und Programme
- 1.3. Entwicklung und Einteilung der Programmiersprachen
- 1.4. Visuelles Programmieren mit Delphi

2. Grundlagen der Programmierung mit Delphi

- 2.1. Die Delphi-Entwicklungsumgebung
- 2.2. Das Prinzip der ereignisgesteuerten Programmierung
- 2.3. Schrittfolge zur Programmerstellung mit Delphi
- 2.4. Projektverwaltung unter Delphi
- 2.5. Struktur einer Unit

3. Delphi und OOP

- 3.1. Vorbemerkungen
- 3.2. Klassen, Instanzen und Vererbung
- 3.3. Objekteigenschaften, Ereignisse und Methoden

Autor und Quellenangaben

Projekte und Übungen

Bitte vorher lesen!

Lineare Algorithmen

- Programmierung eines Taschenrechners
- Berechnung geometrischer Körper

Auswahlstrukturen

- Übungen zu alternativen Strukturen
- Das verrückte Edit-Feld
- Prämienberechnung
- Dreiecksanalyse
- Arbeit mit TListBox-Komponenten

Zyklische Strukturen

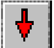
- Einführung - Quadratwurzel nach Heron
- Übungen zu nichtabweisenden Schleifen
- Potenzrechnung
- Fakultät einer natürlichen Zahl

Zeichenketten und strukturierte Datentypen

- Zeichenkettenmanipulator
- Transformationen zwischen Zahlensystemen NEU
- Felder, Such- und Sortierverfahren
- Kryptologische Algorithmen

0. Vorbemerkungen...

... für Informatiklehrer und andere interessierte Kollegen
... für Schüler / Kursteilnehmer
... für Vertreter der Hochschulinformatik
... für Delphi-Programmierfreaks



Für Informatiklehrer und andere interessierte Kollegen:

Bei Weiterbildungen und anderen Treffen erweisen sich die Informatiker unter den Lehrern meist als buntes Völkchen: die einen schwören traditionell auf **Pascal**, andere haben **Visual-Basic** für sich und ihre Schüler entdeckt, wieder andere favorisieren **JavaSkript** u./o. **Java** für den Unterricht, es gibt Vertreter der reinen Lehre des **Oberon.** Und schließlich meinen einige, dass auch **Tabellenkalkulationen**, **Makro- oder Abfragesprachen** eine gangbare Brücke zwischen Schülerinteressen und Lehrplanerfüllung aufspannen. Ach ja, da gibt es noch die Ecken der **C++**-Protagonisten, der Anhänger des deskriptiven **Prolog** und die der **Delphi**-Befürworter und und und.

Das Gemeinsame ist stärker als das Trennende!

Alle haben irgendwo recht und jeder kann triftige Gründe aufführen, die seine Programmierumgebung favorisieren, hat griffige Unterrichtsbeispiele parat, weiß von bemerkenswerten Schülerleistungen zu berichten, hätte gern mehr Zeit und kultusministeriale Protektion für ***sein Fach*** ...

Doch genau letztere steht in Frage, wenn wir bei aller Vielfalt verabsäumen, die uns gemeinsamen Ziele, bezogen auf die Denk- und Arbeitsweisen unserer Schüler an der Schwelle eines Informationszeitalters und den dazu notwendigen Stellenwert unseres Faches in den Mittelpunkt der Diskussion zu stellen - und das reduziert viele scheinbare Unterschiede auf nahe Null.

In diesem Zusammenhang möchte ich allen interessierten und engagierten Fachkollegen die Mitarbeit in den GI-Landesfachgruppen zur Schulinformatik wärmstens ans Herz legen.

Aus obiger Überlegung heraus kann und will ich an dieser Stelle nur auf jene Gründe verweisen, die uns am Lessing-Gymnasium Plauen zum Umstieg von Turbo-Pascal auf Delphi veranlassten ohne zu behaupten, dass andere zeitgemäße Programmiersysteme weniger vorteilhaft einsetzbar sind.

Was spricht also aus unserer Sicht für Delphi und die komponentenbasierte Programmierung?

1. Unter Turbo-Pascal verzeichneten wir eine nachlassende Schülermotivation bezüglich Programmierung. Bezogen auf die gewohnte Welt einer grafischen Benutzeroberfläche erschien das textorientierte Programmiersystem nicht mehr attraktiv genug. Mit Delphi können die Schüler im Handumdrehen eine attraktive und voll funktionierende Benutzeroberfläche für ***ihr*** Programm erstellen.

2. Neben seinen intuitiv bedienbaren visuellen Komponenten (die es natürlich auch für andere Programmiersprachen gibt), besteht Delphi im algorithmischen Kern aus Object-Pascal, dessen Syntax über weite Strecken gleich oder ähnlich Turbo-Pascal ist. Dies ermöglichte einen sanften Übergang.

3. Die visuellen Komponenten entlasten den Schüler von der aufwendigen Programmierung der I/O- und Bildschirmsteuerungsfunktionen und machen sofort den Blick frei für die Umsetzung des eigentlichen Problemlösungsalgorithmus - selbst der gegenwärtig gültige sächsische Lehrplan von '92 kann meines Erachtens damit besser erfüllt werden als mit Turbo-Pascal.

4. Der zum Großteil automatisch vom System erzeugte Quelltext ist durchgehend prozedural gegliedert, so dass die meist in den letzten Lehrplankapiteln thematisierte modulare Programmierung von der ersten Stunde an

integrativer Bestandteil ist, was das Infragestellen des Sinns modularer Strukturen für einfache Programmbeispiele vermeidet.

5. In Delphi wird das Konzept der objektorientierten Programmierung (OOP) verfolgt. Damit hat man im Unterricht einen Zugang zum objektorientierten Softwareentwurf, ist aber dank der visuellen Komponenten keineswegs gezwungen, die Unterrichtsreihe mit einer umfassenden theoretischen Einführung in die OOP zu beginnen.

6. Delphi besitzt eine Datenbankschnittstelle inklusive SQL. In vielen Lehrplänen zur Informatik wird der Umgang mit Datenbanken gefordert oder wahlobligatorisch angeboten. Dabei kann das Einarbeiten in ein spezifisches Datenbankprogramm entfallen, wenn man statt dessen die bis dahin den Schülern vertraute Programmierumgebung von Delphi benutzt.

7. Delphi stellt bezüglich seiner Mächtigkeit als Programmierumgebung vergleichsweise geringe Hardwareanforderungen. So läuft die Version 1.0 unter Windows 3.1 bereits auf einem 486er PC mit 8 MB RAM.

8. Die große Vielfalt an Komponenten und Funktionalität, gepaart mit einer brauchbaren Online-Hilfe eröffnen hervorragende Möglichkeiten zu einer durchgehenden Binnendifferenzierung des Unterrichtes. Begabte Schüler finden massenhaft Anregungen zum selbständigen Erweitern ihrer Projekte.

Was könnte gegen den Einsatz von Delphi sprechen?

1. Der compilierte Programmcode im EXE-Format ist (z.B. im Gegensatz zu Java) plattformabhängig und läuft nur auf Windows-Rechnern.

2. Die Delphi-Programmierumgebung ist für die Schüler nicht kostenlos verfügbar (Hausaufgaben), wenngleich auch zunehmend preiswerte Student-Versionen (EXE-Dateien nicht portierbar!) angeboten werden.

Seitenanfang

Für Schüler / Kursteilnehmer:

Wozu eigentlich programmieren? Braucht man das später wirklich? Es gibt doch soooo viele andere interessante Dinge, die man mit einen Computer machen kann ...

Solche und ähnliche Fragen sind durchaus berechtigt - nur wenige Schüler werden ihr Geld später mit der Programmierung kommerzieller Software verdienen. Doch so gut wie alle werden beruflich und privat mehr oder weniger intensiv mit Computern zu tun bekommen. Keiner kann vorhersagen, welche Anforderungen dann genau vor ihm stehen werden, mit welchen Hard-und Softwarekomponenten er konfrontiert sein wird.

Be**dienen** (des Computers) ist schnell gelernt, weil es von "Dienen" kommt: Mausklick hier, Mausklick da - und wenn nichts mehr geht, wird sich schon einer finden, der ...

Wäre es nicht besser, diese Technik zu **benutzen** oder gar zu **beherrschen**, als nur ihr dienendes Anhängsel zu sein? Computer sollen doch Problemlösungen erleichtern und nicht nur zusätzliche Probleme schaffen! Und genau dabei kann ein Grundverständnis von Programmabläufen, von Objekten, Eigenschaften und Methoden ganz wertvolle Hilfe leisten.

Außerdem vollziehen sich viele Dinge im Alltag in programmähnlichen Strukturen, es müssen Ereignisse vorausgeplant, Problemsituationen analysiert, alternative Strategien gefunden, Aktivitäten vernetzt werden ... Hierbei kann algorithmisches Denken (und das wird beim Programmieren intensiv geschult!) einen vorzüglichen Beitrag leisten, diesen Situationen besser gewachsen zu sein, sie also zu **beherrschen**.

Und nicht zuletzt kann Programmierung auch Spaß machen, spannend und unterhaltsam sein, kreative Kräfte freilegen und immer wieder neu herausfordern - manch einer sprach schon davon, dass man danach fast süchtig werden kann!

So, wer 's nun immer noch nicht gecheckt hat, wozu Programmierung in der Schule gut ist, der gehe bitte zu seinem Mathelehrer und frage ihn, ob er anstelle der Integralrechnung nicht lieber das Ausfüllen und Nachrechnen einer

Steuererklärung in den Mittelpunkt seines Unterrichtesstellen könnte - denn das wird doch garantiert von jedem gebraucht, oder?   ;-)

Seitenanfang

Für Vertreter der Hochschulinformatik:

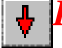
Falls Sie über diese Seiten stolpern und eine wissenschaftlich exakte Einführung in die OOP vermissen, wie sie an Hochschulen / Universitäten gelehrt wird, mag Sie das verdrießen, vielleicht das fachliche Hinterland des Autors in Frage stellen lassen. Letzteres wurde aber durchaus im Studium an der TU Dresden auch mit OOP-Kenntnissen bestellt, allein der Ausgangspunkt und die Voraussetzungen der jeweils Lernenden sind grundverschieden.

Im sächsischen Gymnasium sitzen derzeit Schüler in den Kursen, die in der Mehrheit noch nicht programmiert haben, die aber mehr oder minder mit grafischen Benutzeroberflächen umgehen können und entsprechende Erwartungsbilder an selbst zu erstellende Programme knüpfen. Ergo empfehlen sich Systeme für den Unterricht, die nach dem Baukastenprinzip fertige Oberflächenkomponenten anbieten und den Blick frei machen für die Umsetzung von algorithmischen Grundstrukturen in einer imperativen Sprache. OOP-Kenntnisse entstehen durch den eher intuitiven Umgang mit vorgefertigten Komponenten zunächst nur auf der Stufe einer Propädeutik und werden in ihrer Begrifflichkeit später stufenweise untersetzt und ausgebaut.

Da auch in der Wissenschaft bisher keine allgemein anerkannte Charakterisierung der OOP vorliegt und recht verschiedene Standpunkte vertreten werden, erlaubt sich der Autor in diesem Lehrmaterial folgende didaktische Reduktion:

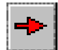
> *OOP im Unterricht reduziert sich zunächst auf die komponentenbasierte Programmierung, geht also von der Existenz vorgefertigter Objekte (Komponenten) aus, deren gegebene Eigenschaften durch zu implementierende Methoden zielgerichtet manipuliert werden. Einführungen in die Begriffswelt der OOP ordnen sich in den jeweils gegebenen Gesamtzusammenhang des Unterrichts ein, der wiederum durch das Problemlösen mittels algorithmischer Grundstrukturen dominiert wird.*

Seitenanfang

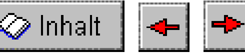


Für Delphi-Programmierfreaks:

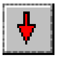
Wenn Ihr an dieser Stelle Bibliotheken mit tollen, nie da gewesenen Komponenten der neuesten Delphi-Version oder Sammlungen von Tipps und Tricks auf Expertenebene erwartet, dann seit ihr hier leider falsch!
Und wenn Ihr in meinen Seiten dennoch einmal blättert, dürfte Euch Umfang und Schwierigkeitsgrad der hier vorgestellten Projekte allenfalls ein müdes Lächeln abgewinnen, das durch mein Eingeständnis, im Jahre 1999 noch mit der Version 1.0 zu programmieren, in schallendes Gelächter übergehen wird. Wie dem auch sei, bedenkt bitte eines: Die meisten unserer Elftklässler haben bezüglich Programmierung einen Vorkenntnisstand nahe Null, und es ist auch nicht erklärtes Ziel der Informatikkurse, in knapp 60 Stunden "fertige" Programmierer auszubilden. Grundlagen, typische Denk- und Arbeitsweisen sowie die Befähigung zum selbständigen Weiterlernen sind mir tausendmal wichtiger als hochkarätige Softwareprojekte auf neuester Plattform. Lieber sollen sich bekennende Freaks rechtzeitig outen, aus dem regulären Unterricht ausklinken und eigene Projekte durchziehen - und dabei kamen schon äußerst bemerkenswerte Programme heraus!
PS: auch der Autor dieser Seiten hat schon mehr programmiert als nur einen Taschenrechner ;-)



1.Einführung in Delphi



1.1. Was ist Delphi?
1.2. Algorithmen und Programme
1.3. Entwicklung und Einteilung der Programmiersprachen
1.4. Visuelles Programmieren mit Delphi



1.1. Was ist Delphi?

Delphi ist ein Entwicklungssystem zum Erstellen von Windowsprogrammen. Dazu werden ein leistungsfähiger Pascal-Compiler, visuelle Komponenten und die Möglichkeit des Erstellens von Datenbankprogrammen in einem System vereinigt.

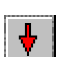
Mit Delphi kann *jeder* einfach, sicher und schnell Windowsprogramme entwickeln.

Der Vorteil von Windowsprogrammen liegt in ihrer einheitlichen Bedienung. Die meisten Windowsprogramme besitzen eine Menüleiste und ein Hauptfenster und lassen sich größtenteils mit der Maus bedienen. Programme werden in Fenstern ausgeführt, die oft nur einen Teil des gesamten Bildschirmes beanspruchen. Dieser Fenstertechnik verdankt Windows seinen Namen. Über Fenster und Dialoge, die sogenannten Benutzerschnittstellen, kommuniziert der Anwender mit dem Programm.

Die grafische Benutzeroberfläche von Windows unterscheidet sich in vielen Punkten vom textorientierten Betriebssystem DOS. Dies hat auch für den Programmierer Konsequenzen.

DOS	WINDOWS
Unter DOS kann nur ein Programm gestartet werden.	Mehrere Programme können gleichzeitig gestartet werden.
Das laufende Programm besitzt alleinigen und uneingeschränkten Zugriff auf die Hardware.	Mehrere Programme können gleichzeitig auf die Hardware zugreifen, z.B. auf den Drucker. Der Zugriff auf die Hardware wird von Windows kontrolliert.
Die Hardware wird direkt programmiert.	Windows stellt Funktionen für den Zugriff auf die Hardware zur Verfügung. Es werden z.B. dieselben Funktionen zur Druckeransteuerung für Laser-, Tintenstrahl- und Nadeldrucker verwendet.
Die Oberfläche ist textorientiert.	Die Oberfläche ist grafikorientiert.
Das Programm wartet auf Benutzereingaben, indem es in einer Schleife die Maus und die Tastatur abfragt.	Windowsprogramme bekommen eine Nachricht, wenn für sie eine Maus- oder Tastatureingabe vorliegt.
Die Ein- und Ausgabe ist bildschirmorientiert.	Die Ein- und Ausgabe ist fensterorientiert.

Seitenanfang



1.2. Algorithmen und Programme

Die elektronische Datenverarbeitung (EDV) ermöglicht die Erleichterung der menschlichen Arbeit durch den Einsatz von Maschinen bzw. Computern. Damit der Computer verschiedene Arbeitsschritte automatisch ausführen kann, müssen diese vorher genau beschrieben werden. Ein Algorithmus ist eine Folge von Anweisungen, die genau diese Arbeitsschritte beschreiben.

Allgemein bezeichnet man einen Algorithmus als eine eindeutige Beschreibung eines endlichen Verfahrens zur Lösung einer Vielzahl von Problemen gleicher Art.

Jedes Problem, dessen Lösung durch einen Algorithmus beschrieben werden kann, ist im Prinzip durch einen Computer lösbar.

Ein Programm ist eine Folge von Anweisungen (Algorithmus), die in einer Programmiersprache wie z.B. Pascal formuliert sind.

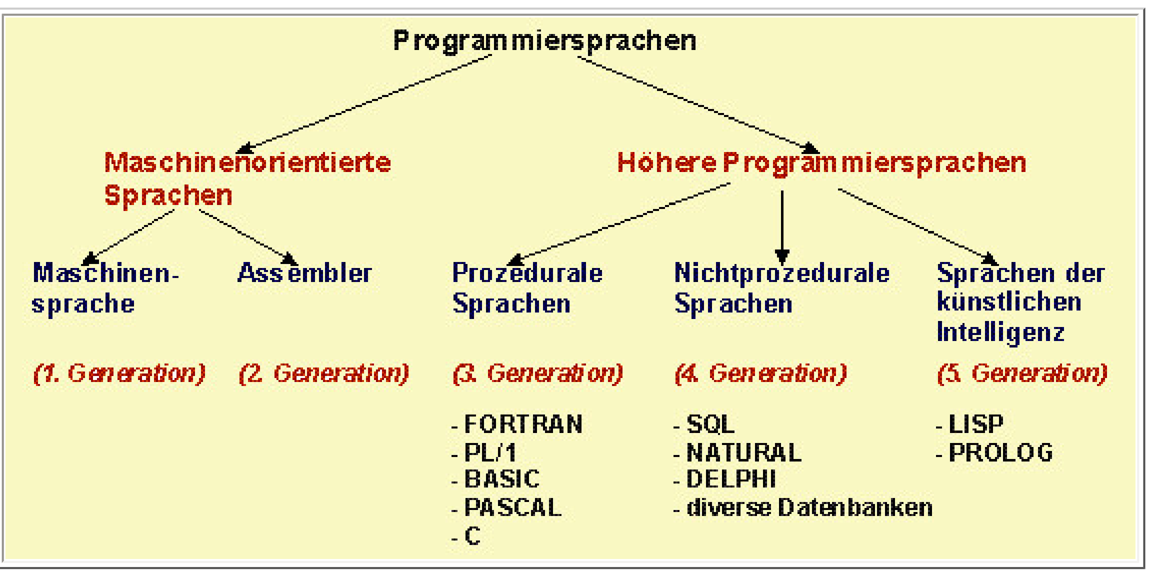
In einem Programm stehen somit nur Anweisungen, die der Computer versteht und umsetzen kann.

Seitenanfang

1.3. Entwicklung und Einteilung der Programmiersprachen

Da man noch nicht in natürlicher Sprache mit einem Rechner kommunizieren kann, wurden im Laufe der Jahre verschiedene Programmiersprachen entwickelt.
Der Unterschied zwischen gesprochenen Sprachen und Programmiersprachen liegt darin, dass die Worte einer Programmiersprache nur eine Bedeutung zulassen, während der Sinngehalt mancher Worte der Umgangssprache erst aus dem Kontext heraus deutlich werden kann. Ein Rechner benötigt aber stets eindeutig formulierte Anweisungen zur Bearbeitung.

Wie viele Programmiersprachen und Programmiersysteme es heute weltweit gibt, lässt sich nicht beantworten. Es können einige hundert sein, da viele Sprachen nur für spezielle Aufgaben und Einsatzgebiete konzipiert wurden. Die bekanntesten Programmiersprachen lassen sich in Auszügen in folgende Hauptgruppen unterteilen:



1. Generation: Maschinensprachen

Die ersten EDV-Anlagen (Ende der 40er Jahre) ließen sich nur maschinennah programmieren. Der Programmcode musste bitweise in den Speicher des Rechners geschrieben werden. Der Vorteil der maschinennahen Programmierung liegt bis heute darin, dass diese Art von Programm direkt von einem Computer ausgeführt werden kann. Allerdings sind sehr genaue Rechnerkenntnisse erforderlich, da alle Anweisungen in Form von elementaren Befehlen sehr kleinschrittig beschrieben werden müssen. Problematisch gestaltet sich die Fehlersuche, wenn ein Programm überhaupt nicht läuft oder falsche Ergebnisse liefert.

Beispiel:

```
11001011   11100011
00110101   10111101
```

2. Generation: Assemblersprachen

Die Assemblersprachen, deren Befehlsvorrat speziell für jeden Rechnertyp zugeschnitten ist, verwenden anstelle des Binärcodes leichter verständliche Symbole, Mnemonics genannt. Ein Assemblerprogramm ist auf einem Computer nicht mehr direkt ablauffähig, sondern muss erst in ein entsprechendes Maschinenprogramm übersetzt werden. Ein Programm, das dies automatisch durchführt, bezeichnet man als Assembler, den Übersetzungsvorgang als assemblieren. Der Nachteil von Assemblerprogrammen besteht darin, dass sie auf eine ganz bestimmte Hardware zugeschnitten sind und sich nur schwer auf andere Computertypen übertragen lassen. Bei größeren Problemlösungen werden die Programme sehr umfangreich und damit wartungsunfreundlich. Daher werden Assemblersprachen hauptsächlich nur noch da, wo Programme und Programmsysteme schnell reagieren müssen, und für Teile des Betriebssystems eingesetzt.

Beispiel:

```
ADD FELD_2 FELD_3
MOV BX, OFFSET FELD_3
```

3. Generation: Prozedurale Programmiersprachen

Diese Sprachengeneration, der die überwiegende Mehrheit der heute gebräuchlichen Programmiersprachen angehört, ist unabhängig von einem Computersystem. Lediglich der Übersetzer (Interpreter oder Compiler) muss an das jeweilige System angepasst sein und den entsprechenden Maschinencode erzeugen. Prozedurale Sprachen besitzen einen speziellen, der menschlichen Sprache angenäherten Befehlssatz, um Probleme aus einem bestimmten Anwendungsbereich zu lösen. Sie lehnen sich somit an die Denkweise des Programmierers an. Auch ohne fundamentierte Programmierkenntnisse lassen sich diese Programme leicht nachvollziehen. Die Bezeichnung "prozedural" kennzeichnet den modularen Aufbau der entsprechenden Programme in Prozeduren oder Funktionen.

Beispiel:

```
Write('Fahrstrecke='); Readln(kilometer);
Write('Benzin='); Readln(liter);
verbrauch := liter/kilometer * 100;
Writeln('Sie verbrauchten auf 100 km ',verbrauch);
if verbrauch > 7 then writeln "Verbrauch zu hoch!";
```

4. Generation: Nichtprozedurale Programmiersprachen

Bei nichtprozeduralen Programmiersprachen wird nicht mehr festgelegt, **wie** ein Problem gelöst wird, sondern der Programmierer beschreibt lediglich, **was** das Programm leisten soll. Danach werden diese Angaben von dem Programmiersystem in ein Programm umgesetzt. Der Vorteil dieser Sprachen besteht darin, dass für diese Art der Programmierung keine umfangreiche Programmierausbildung notwendig ist. Nichtprozedurale Programmiersprachen werden z.B. für Datenbankabfragen oder Tabellenkalkulationen eingesetzt.
In **Delphi** verwendet man z.B. die visuellen Komponenten, um eine Benutzerschnittstelle zu erstellen.

Beispiel:

```
select KUNDE from TABLE_1 where ALTER > 18
create ERWACHSENE
```

5. Generation: Programmiersprachen der künstlichen Intelligenz

Die Programmierung der künstlichen Intelligenz (KI) dient der fortgeschrittenen Programmierung. Es wird versucht, die natürliche Intelligenz des Menschen (z.B. seine Lernfähigkeit) durch entsprechend konstruierte Computer nachzuvollziehen. Hierbei fließt beispielsweise auch die natürliche Sprache in die Programmierung ein. KI-Programme werden überwiegend zu Forschungszwecken eingesetzt und beschreiben Schlussfolgerungen aus Forschungsergebnissen. Erfolgreich werden derartige Systeme zur Spracherkennung eingesetzt.

Beispiel: `Berechnung auswerten.`

Einordnung von Delphi:

Als komplexes Programmiersystem lässt sich Delphi in zwei Generationen einordnen:

- **die von Delphi verwendete Programmiersprache Object Pascal ordnet man in der 3. Generation ein.**
- **die visuellen und SQL-Komponenten gehören der 4. Generation an.**

Aufgrund dieser Einordnung wird Delphi auch als eine **hybride Programmiersprache** bezeichnet.
Die Kombination der Funktionalität zweier Generationen von Programmiersprachen mit visuellen Programmiertechniken führen zu einer hohen Bedienerfreundlichkeit bei der Programmerstellung gepaart mit einer enormen Mächtigkeit der erzeugbaren Programme.

Seitenanfang

1.4. Visuelles Programmieren mit Delphi

Interpreter und Compiler

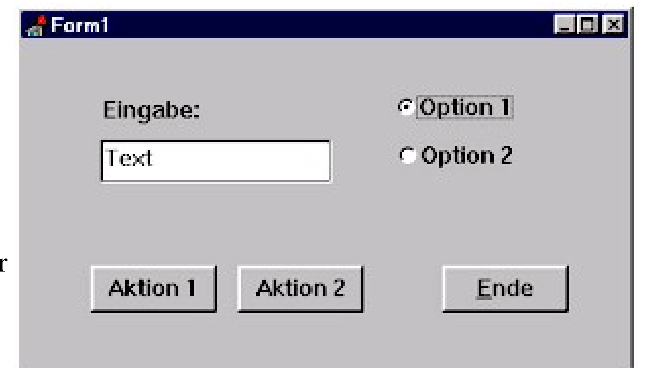
Der Prozessor eines Computers kann nur Maschinenbefehle lesen (bestehend aus Binärcode 0/1). Programme, die nicht in Maschinensprache geschrieben sind, müssen erst in diese übersetzt werden.
Die Aufgabe des Übersetzens übernehmen eigens dafür entwickelte Programme, Interpreter oder Compiler genannt.

Interpreter	Interpreter übersetzen (interpretieren) die Programme zeilenweise. Das Programm kann deshalb zur Laufzeit geändert werden. Die Befehle werden Zeile für Zeile in Maschinensprache übersetzt und vom Prozessor ausgeführt. Bei jedem Neustart des Programms muss dieses auch wieder neu interpretiert werden. Aus diesem Grund können keine Optimierungen vorgenommen werden, und die Programme laufen langsamer ab. *Beispiele für Interpreter-Sprachen:* Q-BASIC, JAVA, LOGO
Compiler	Ein Compiler übersetzt einen Programmtext vollständig in Maschinensprache und legt diesen in einer eigenständigen Programm-Datei ab. Während der Compilierung optimiert der Compiler die Programmgröße und -geschwindigkeit. Beim Neustart wird vom Prozessor direkt die Programmdatei abgearbeitet. Dadurch laufen compilierte Programme 10 bis 20 mal schneller ab als zu interpretierende Programme. *Beispiele für Compiler-Sprachen:* PASCAL, DELPHI, C++

Visuelles Programmieren

Delphi erleichtert durch seine visuellen Komponenten wie Menüs, Schaltflächen und Oberflächenkomponenten das Erstellen einer Benutzerschnittstelle in Windows. Dadurch wird die Komplexität der Windowsprogrammierung, die auf Fenstern und Botschaften beruht, wesentlich vereinfacht.
Hinter den visuellen Komponenten verbergen sich nicht nur grafische Darstellungen. Vielmehr stellt jede Komponente dem Programm eine oder mehrere Funktionen zur Verfügung.

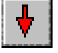
Das Programmieren unter Windows baut auf zwei wichtigen Konzepten auf, den **Fenstern** und den **Botschaften**.



Die Abbildung zeigt ein typisches **Dialogfenster**, welches mit visuellen Komponenten in Sekundenschnelle und ganz ohne "Insiderkenntnisse" erstellt werden kann.
Der Anwender kommuniziert über dieses Fenster mit dem jeweiligen Programm. Im Eingabefeld kann ein beliebiger Text oder Zahlenwert eingegeben werden, der dann über das Anklicken eines Aktionsschalters vom Programm verarbeitet wird.
Dabei stellt das Ereignis *"Schalter geklickt"* eine **Botschaft** an das Programm dar, woraufhin dieses einen entsprechenden Algorithmus (Ereignisbehandlungsroutine) ausführt.

2. Grundlagen der Programmierung mit Delphi

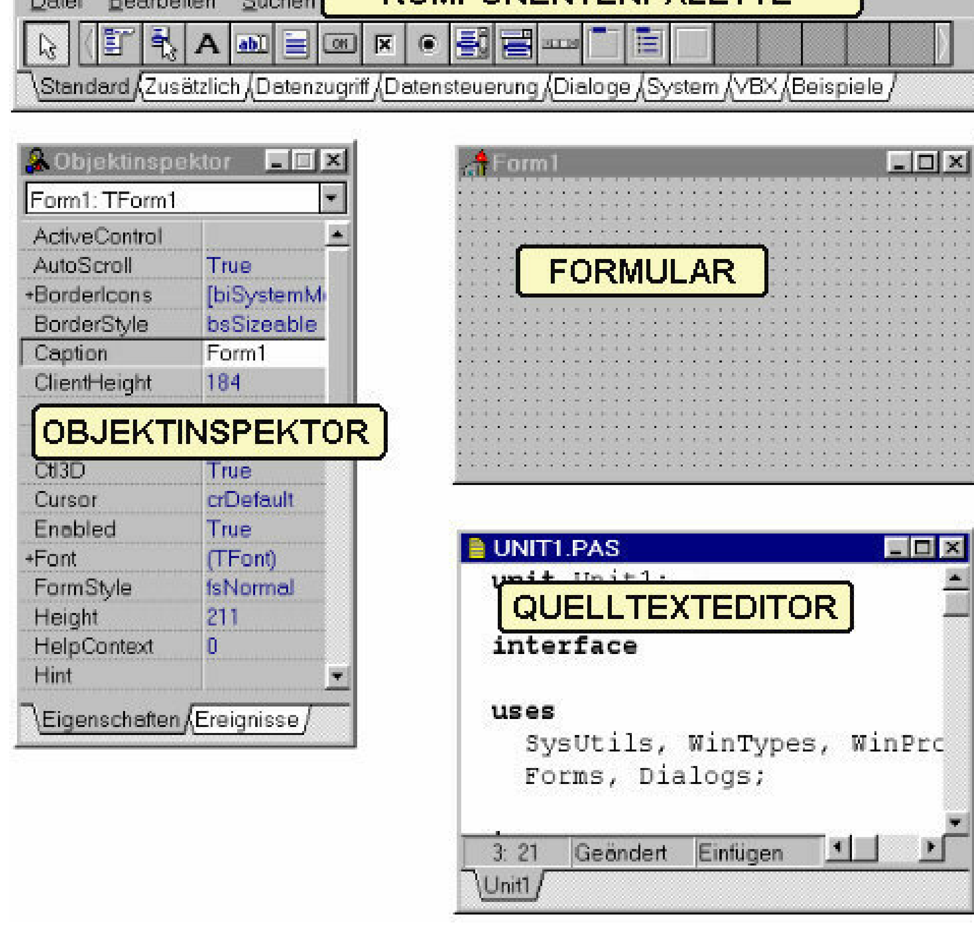
- 2.1. Die Delphi-Entwicklungsumgebung
- 2.2. Das Prinzip der ereignisgesteuerten Programmierung
- 2.3. Schrittfolge zur Programmerstellung mit Delphi
- 2.4. Projektverwaltung unter Delphi
  - 2.4.1. Beispiel für die Dateistruktur eines Projektes
  - 2.4.2. Dateien, die in der Entwicklungsphase erzeugt werden
  - 2.4.3. Vom Compiler erzeugte Projektdateien
  - 2.4.4. Empfohlene Vorgehensweise im Unterricht - Projektsicherungsprogramm
- 2.5. Struktur einer Unit



2.1. Die Delphi-Entwicklungsumgebung

Nach dem Start von Delphi werden verschiedene Fenster geöffnet, die für die visuelle Programmierarbeit notwendig sind. Diese Arbeitsfläche wird als Integrierte Entwicklungsumgebung, kurz IDE (engl. Integratet Development Enviroment) bezeichnet. Alle Fenster der IDE können frei und einzeln auf dem Bildschirm positioniert oder geschlossen werden. Durch das Schließen des Hauptfensters wird Delphi beendet.

Die nachstehende Tabelle gibt einen einführenden Überblick zu Erscheinungsbild und Funktion der wichtigsten Bestandteile der Delphi-Entwicklungsumgebung:



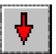
Das **Formular** stellt das zentrale Entwicklungsobjekt eines Delphi-Programms dar. Auf ihm werden die gewünschten Komponenten wie Schaltflächen, Menüs und Eingabefelder per Mausklick platziert und größenmäßig angepasst. Das Erscheinungsbild des Formulars entspricht dem Aussehen des Windows-Fensters, in dem das fertige Programm später ablaufen wird.

Die **Komponentenpalette** ist in verschiedene Register (Standard, Zusätzlich usw.) unterteilt, und diese erlauben die Auswahl der benötigten Windows-Komponenten.

Mit Hilfe des **Objektinspektors** werden Darstellungsweise und Verhalten der Objekte (Komponenten) in einem Formular sowie das Aussehen des Formulars selbst festgelegt. Das Erscheinungsbild wird über die Seite "Eigenschaften", das Verhalten beim Eintritt eines Ereignisses über die Seite "Ereignisse" eingestellt.

Der **Quelltexteditor** wird zum Schreiben des Programmcodes in der Programmiersprache Pascal genutzt. Dabei generiert Delphi für neu in das Formular eingefügte Komponenten automatisch den Programmcode, der dann im Editor angezeigt wird. Dem Programmierer obliegt daher "nur noch" die Einarbeitung der Algorithmen zur Ereignisbehandlung.

Seitenanfang



2.2. Das Prinzip der ereignisgesteuerten Programmierung

Die Programmierung mit Delphi wird auch als ereignisgesteuerte Programmierung bezeichnet. Ein Ereignis ist eine Aktion, die den Programmablauf beeinflusst.

Was sind Ereignisse?

Alle Aktionen (Tastatureingaben, Mausbewegungen) eines Benutzers, wie zum Beispiel:

- Einfaches oder doppeltes Klicken auf eine Befehlsschaltfläche,
- Verschieben, Öffnen oder Schließen eines Fensters mit der Maus,
- Positionieren des Cursors in ein Eingabefeld mit der Tabulatortaste.

Interne Programmabläufe, wie zum Beispiel:

- Berechnungen durchführen,
- Öffnen und Schließen eines Fensters (vom Programm gesteuert),
- Ermitteln von aktueller Uhrzeit und Datum.

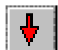
Reaktionen auf Ereignisse

Mit Delphi werden Programme entwickelt, die durch die grafische Benutzeroberfläche von Windows mit Steuerelementen wie beispielsweise Dialogfenstern, Schaltflächen und Eingabefeldern gesteuert werden.

Durch ein **Ereignis** (z.B. Klicken auf eine Schaltfläche) wird eine **Reaktion** (z.B. Öffnen eines Fensters) hervorgerufen. Diese Reaktion wird in Form von Object-Pascal-Code in einer Prozedur erstellt, die als **Ereignisbehandlungsroutine** bezeichnet wird.

Als Ereignisbehandlungsroutinen werden diejenigen Anweisungen bezeichnet, die ausgeführt werden, sobald ein Ereignis eintritt.

Seitenanfang



2.3. Schrittfolge zur Programmerstellung mit Delphi

Die nachfolgende Tabelle soll aufzeigen, wie sich der "klassische" Software-live-cycle unter den Gegebenheiten einer visuellen Programmierumgebung erweitert bzw. modifiziert. Dabei darf man sich den Durchlauf der einzelnen Schritte keinesfalls als einmalige lineare Abfolge vorstellen, vielmehr entstehen durch Rücksprünge und das mehrmalige Durchlaufen von Teilen der Schrittfolge zyklische Strukturen. Werden z.B. bei der Resultatsprüfung unter Punkt 5) Fehler festgestellt, muss wieder bei Punkt 2A) begonnen werden, indem man logische Fehler im Algorithmus beseitigt.

Und schließlich kann auch die gesamte Schrittfolge als zyklische Struktur begriffen werden: kaum ist die Version 1.0 eines Programms ausgeliefert, werden bisher unerkannte "Bugs" entdeckt und Wünsche zur Erweiterung der Funktionalität des Programms laut - also neue Problemformulierung, Problemanalyse, ... und schon existiert die Version 1.1 usw.

1) Problemformulierung

"Es ist ein Programm zu erstellen, das folgendes leistet: ... "
So oder so ähnlich beginnt die Formulierung dessen, was das Programm im ganzen realisieren soll. Man wird mehr oder weniger klare Vorstellung zu notwendigen Eingabewerten, zum Erscheinungsbild des Programms, zur Art der Ergebnisausgabe, zum potentiellen Nutzerprofil etc. vorfinden.

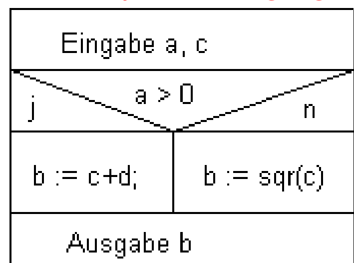
Die nachfolgenden beiden Schritte verlaufen im Prinzip parallel - die Programmierhandlung springt zwischen algorithmischer und visueller Seite hin und her, weil der algorithmische Aufbau des Programms bestimmte Komponenten einer Nutzeroberfläche bedingt bzw. die gewünschte Oberflächenstruktur Einfluss auf die Modularisierung der Lösungsalgorithmen haben kann.

2A) Algorithmischer Problemlösungsprozeß - Algorithmische Seite -

I) Problemanalyse

Das Gesamtproblem wird in überschaubare Teilprobleme zerlegt - Modularisierung und Modellierung der Problemsituation.

II) Entwurf von Lösungsalgorithmen für die Teilprobleme

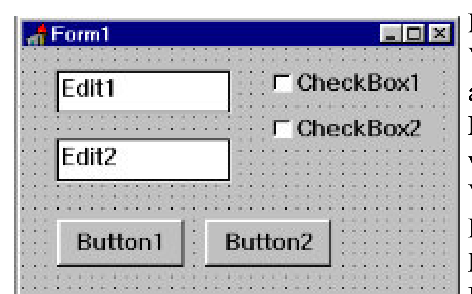


Die Lösungswege dazu werden zunächst in abstrakter Weise beschrieben. Je nach Komplexität kann dies in verbaler Form geschehen (Eindeutigkeit!) oder es kommen grafische Beschreibungsformen, z.B. Struktogramme oder Programmablaufpläne zum Einsatz. Es wird noch nicht mit konkreter Programmiersprache gearbeitet.

III) Synthese der Teillösungen

Nunmehr werden die entwickelten Teillösungen zur Gesamtlösung verknüpft und notwendige E/A-Funktionen zwischen den einzelnen Modulen festgelegt.

2V) Benutzeroberfläche erstellen und Eigenschaften der Objekte festlegen - Visuelle Seite -



Mit zunehmenden Voranschreiten im algorithmischen Problemlösungsprozeß wächst auch die Vorstellung über Beschaffenheit und Funktionalität der Benutzeroberfläche.

Erstellt wird diese durch Anordnung aller notwendigen Komponenten (Textfelder, Optionsfelder, Eingabefelder, Schalter usw.) auf dem Formular.

Die Auswahl der Komponenten erfolgt über die Komponentenpalette. Dabei werden mit Hilfe des Objektinspektors die Eigenschaften der Komponenten festgelegt, z.B. Größe, Farbe, Bezeichnung der Schaltflächen, Text- und Eingabefelder.

Spätestens an dieser Stelle sollte das Projekt erstmals gespeichert werden!

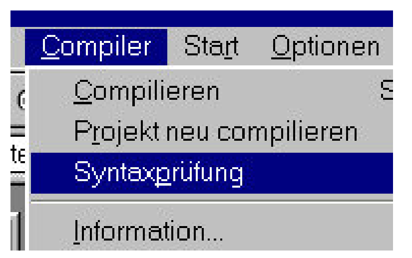
4) Ereignisbehandlung codieren

- "Hochzeit" von Algorithmus und Programmoberfläche -

Zunächst werden über den Objektinspektor diejenigen Ereignisse ausgewählt, die für den späteren Programmablauf von Bedeutung sein werden (z.B. Klicken eines Schalters).
In einer zugehörigen Ereignisbehandlungsroutine wird dann im Quelltexteditor in der Programmiersprache Pascal beschrieben, wie die jeweiligen Komponenten auf das betreffende Ereignis reagieren sollen.

Dabei werden also die unter 3A) gefundenen Lösungsalgorithmen in Programmiersprache "übersetzt" und die Ergebnisausgabe wiederum über Komponenten realisiert.

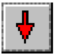
5) Test-Korrektur-Durchläufe

I) Syntaxprüfung:	Vorm ersten Start sollte unbedingt die syntaktische Korrektheit der eingegebenen Pascal-Anweisungen getestet und ggf. korrigiert werden. Anschließend wird das Projekt erneut gespeichert!	
II) Resultatsprüfung:	Das syntaktisch korrekte Programm wird nun compiliert und gestartet. Um die Richtigkeit seiner Resultate hinreichend abzusichern, muss es mit verschiedenen Beispieleingaben getestet werden. Treten während des Tests Fehler auf, sind die oben genannten Schritte zu wiederholen.	

6) Sicherung der Dateien, Dokumentation und Nutzung des Programms

In einem eigens dafür angelegten Verzeichnis (z.B. auf Diskette) werden abschließend alle für eine spätere Weiterbearbeitung des Programmprojektes nötigen Dateien gesichert.
Die beim Compilieren entstandene Programmdatei kann nunmehr unabhängig von Delphi unter Windows genutzt werden.

Seitenanfang



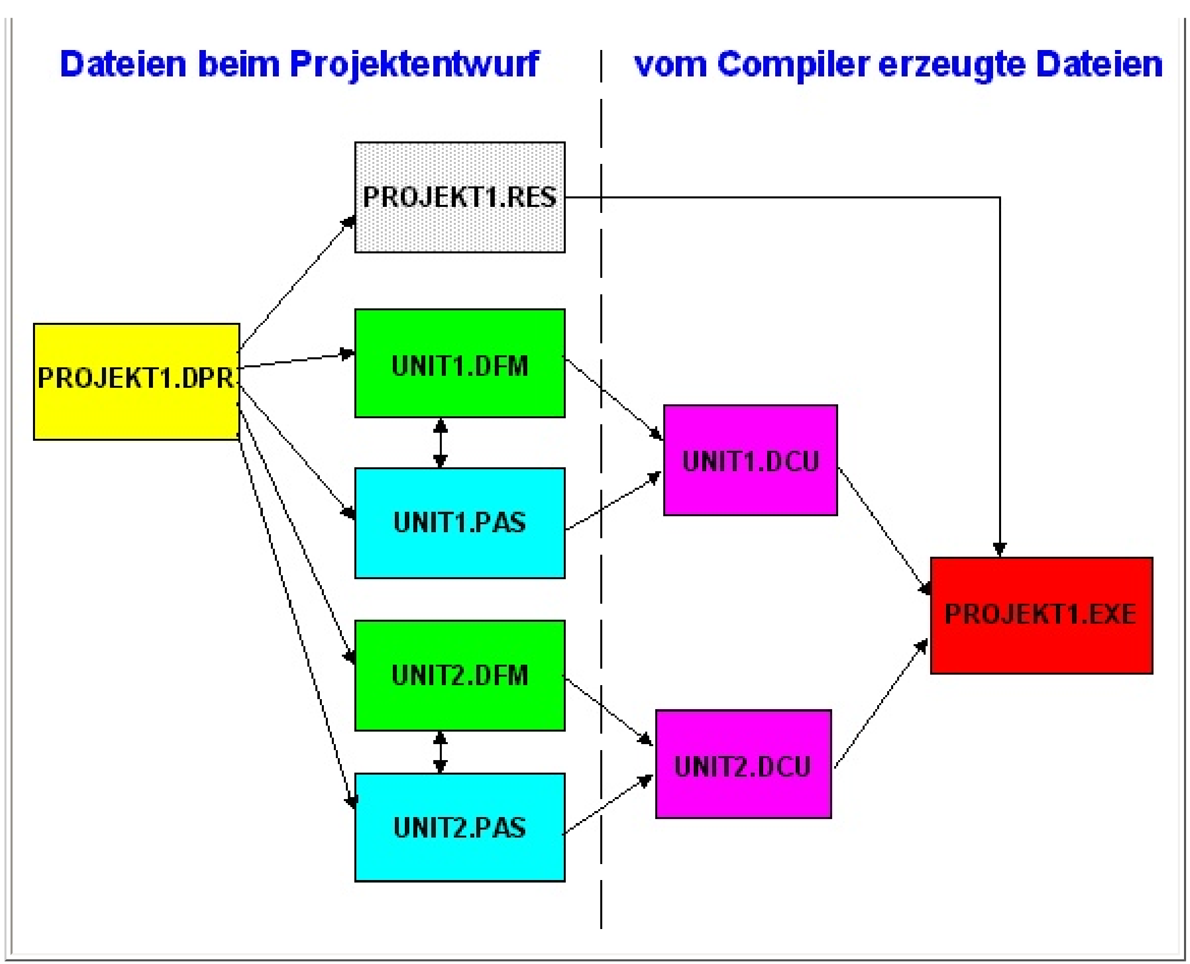
2.4. Projektverwaltung unter Delphi

Ein Delphi-Projekt ist eine Sammlung aller Dateien, die zusammen eine Delphi-Anwendung auf dem Entwicklungssystem ergeben. Einige dieser Dateien werden im Entwurfsmodus erzeugt, andere werden bei der Compilierung des Projekt-Quelltextes angelegt.

> ***Merke: Jedes Projekt sollte unbedingt in einem separaten Verzeichnis gespeichert werden.***

2.4.1. Beispiel für die Dateistruktur eines Projektes

Ein Delphi-Projekt namens Projekt1 bestehe aus zwei Formularen: (Dateien sind zur Erläuterung anklickbar)



2.4.2. Dateien, die in der Entwicklungsphase erzeugt werden:

Dateinamen-erweiterung	Definition	Zweck / Bemerkungen
.DPR	***Projektdatei***	❍ Pascal-Quelltext für die Hauptprogrammdatei des Projektes, ❍ enthält den Standardnamen des Projektes, ❍ verzeichnet alle Formular- und Unit-Dateien im Projekt und enthält den Initialisierungscode ❍ **wird von Delphi verwaltet und sollte nicht manuell verändert werden!**
.DFM	***Grafische Formulardatei***	❍ Quelltext, der die Entwurfseigenschaften eines Formulars des Projekts enthält, ❍ für jedes projektzugehörige Formular wird beim ersten Speichern des Projekts eine .DFM Datei zugleich mit der entsprechenden .PAS Datei angelegt. ❍ **Datei selbst wird von Delphi verwaltet während die Entwurfseigenschaften des Formulars vom Programmierer über den Objektinspektor eingestellt werden!**
.PAS	***Unit-Quelltext*** (in Pascal)	❍ wichtigste Bausteine für den Programmablauf! ❍ Für jedes Formular wird automatisch eine zugehörige Unit erzeugt, die alle Deklarationen und Prozeduren (Methoden) für die vom Formular auslösbaren Ereignisse enthält. ❍ **nur in diese Dateien wird vom Programmierer der eigentliche Programmquelltext geschrieben!**
.RES	***Compiler-Ressourcendatei***	❍ Binärdatei für vom Projekt zu verwendende äußere Ressourcen ❍ **wird von Delphi verwaltet**

Zur Datensicherung bzw. zur Speicherung von Systemeinstellungen erzeugt Delphi noch weitere Dateiarten, die hier nicht aufgeführt sind.

Um ein Projekt jedoch zwecks späterer Weiterbearbeitung zu sichern, genügt es (für den Einsteiger), alle zum Projekt gehörenden Dateien mit den oben aufgeführten Erweiterungen zu sichern.

2.4.3. Vom Compiler erzeugte Projektdateien:

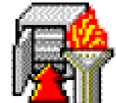
Dateinamen-erweiterung	Definition	Zweck / Bemerkungen
.DCU	*Unit-Objekt-Code*	○ während der Compilierung wird automatisch aus jeder .PAS-Datei und der zugehörigen .DFM-Datei eine entsprechende .DCU Datei erzeugt, die alle Eigenschaften und Methoden eines Formulars im Binärcode enthält.
.EXE	*Compilierte ausführbare Programmdatei*	○ vertriebsfähige Programmdatei, die alle für das Programm nötigen .DCU-Dateien enthält.

Soll also das fertige Delphi-Programm auf andere Rechner übertragen werden, genügt es, die entsprechende .EXE Datei dorthin zu kopieren. Dabei müssen diese Rechner natürlich unter Windows laufen und die nötigen Delphi-Ressourcen besitzen.

2.4.4. Empfohlene Vorgehensweise im Unterricht:

Das nachfolgende Szenario geht davon aus, dass der Unterricht in einem vernetzten Computerkabinett durchgeführt wird, wobei jedem Schüler ein temporär gemapptes Serverlaufwerk (bei uns L:) zum Lesen und Schreiben zur Verfügung steht. Auf diesem Laufwerk werden zur Entwicklungszeit alle zum Delphi-Projekt gehörenden Dateien geführt.

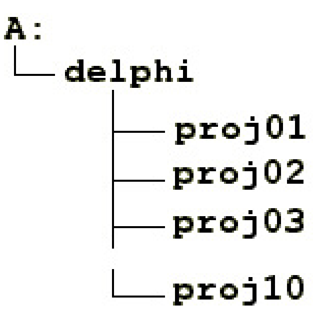
Zum Unterrichtsende speichern die Schüler alle Projektdateien dauerhaft auf Diskette, und falls in der folgenden Stunde am Projekt weiter gearbeitet werden soll, werden die notwendigen Dateien wiederum von A: nach L: kopiert.
Ein ausschließliches Arbeiten mit Disketten hat sich als wenig praktikabel erwiesen, da das Compilieren dann jedes Mal unangenehm lange dauert und durch das dauerhafte Ablegen der relativ großen EXE-Dateien die Disketten schon nach wenigen Projekten voll sind.



Um die Speicherung auf die wirklich notwendigen Projektdateien zu begrenzen, wurde (übrigens auch unter Delphi!) ein einfaches Projektsicherungsprogramm namens projsave.exe entwickelt und steht hier zum Download bereit. Eventuelle Nutzer müssten natürlich das beschriebene Szenario den Gegebenheiten in ihrem Kabinett anpassen ;-)
Wer das Programm modifizieren oder verbessern möchte, kann sich auch die Quelltexte im Zip-Format herunter laden - Mail an mich wäre nett!

Für das derzeitige Netzwerk des Lessing-Gymnasiums Plauen sieht das ganze dann so aus:

Man lege sich auf Diskette im Laufwerk A: eine geeignete Verzeichnisstruktur an!

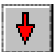


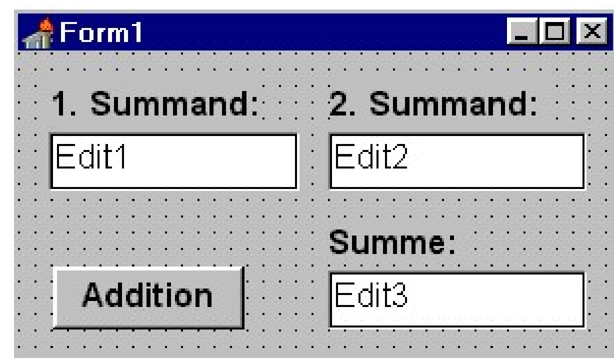
Ein neu zu erstellendes Projekt wird während seiner Bearbeitung auf Laufwerk **L:** (Server) zwischengespeichert. Nachdem Delphi beendet ist, verwende man das Programm **„PROJSAVE.EXE“** und stelle als Quelle L: und als Ziel ein freies Verzeichnis auf **A:** ein (z.B. A:\delphi\proj02).

Das **Sicherungsprogramm** ist so konzipiert, dass es nur die für eine spätere Weiterbearbeitung wirklich notwendigen Dateien auf A: kopiert und damit sparsam mit dem Speicherplatz der Diskette umgeht. Der gesamte Inhalt des (temporären!) Laufwerks L: wird in dem Moment automatisch gelöscht, da der Schüler sich vom Server abmeldet. Nicht gesicherte Dateien sind spätestens dann unweigerlich verloren!

Soll zu einem späteren Zeitpunkt das Projekt weiterbearbeitet werden, sorge man zunächst dafür, dass Laufwerk **L:** von evtl. vorhandenen Delphi-Dateien aus anderen Projekten bereinigt wird (über „PROJSAVE.EXE“ möglich). Danach kopiere man sich über das Sicherungsprogramm die zu bearbeitenden Delphi-Dateien von **A:** nach **L:**, und nachdem Delphi gestartet wurde, kann deren Bearbeitung weitergehen.

Seitenanfang

 2.5. Struktur einer Unit unter Delphi



Die nachfolgend aufgelistete Unit bezieht sich auf das abgebildete einfache Formular - Addition zweier Zahlen.
Bemerkenswert (und beruhigend!) ist die Tatsache, dass alle in der Unit **schwarz geschriebenen Zeilen** während der visuellen Formularerstellung **automatisch von Delphi erzeugt** werden.
Nur die wenigen **rot gekennzeichneten Zeilen** der Prozedur zur Realisierung der Addition sind wirklich **vom Programmierer zu tippen**.
Dies untersetzt die eingangs getroffene Feststellung, dass auch Programmier-Einsteiger ohne nennenswerte Vorkenntnisse unter Delphi schnell zu "greifbaren" Ergebnissen - sprich ablauffähigen Programmen - gelangen und einfache Algorithmen implementieren können. Es erscheint daher sinnvoll, erst nach einigen Projektübungen zum praktischen Teil der Programmierung die Lernenden sukzessive mit der Struktur einer Unit und der Funktion ihrer Bestandteile vertraut zu machen.

Strukturelemente der Beispiel-Unit	Benennung / Bedeutung
unit Utest1;	**Kopfzeile der Unit:** enthält deren Dateinamen
interface	**Interface-Teil**: bestimmt, was in der Unit von außen zugänglich ist
uses SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls, Forms, Dialogs, StdCtrls;	**Uses-Teil**: Benennung von Units (Prozedurbibliotheken), die von der aktuellen Unit verwendet werden

type
TForm1 = **class**(TForm)
  Edit1: TEdit;
  Edit2: TEdit;
  Edit3: TEdit;
  Label1: TLabel;
  Label2: TLabel;
  Label3: TLabel;
  Button1: TButton;
  procedure Button1Click(Sender: TObject);
end;
	Typdeklaration des Formularobjektes:
	
	enthält alle Komponenten, die auf dem Formular angeordnet sind sowie die zum Formular gehörenden Prozeduren

var **Form1: TForm1;**	**Deklaration globaler Variablen** hier: Variable (Objekt) Form1 vom Typ TForm1 (Objekttyp)
implementation	**Implementationsteil**: enthält Programmteil der Prozeduren
*{$R *.DFM}*	**Formulardatei wird an die Unit gebunden**
procedure TForm1.Button1Click(Sender: TObject);	**Kopfzeile der Prozedur**
var a, b, c: real;	**Deklaration lokaler Variablen:** diese gelten nur in der jeweiligen Prozedur

begin
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  c := a+b;
  Edit3.Text := FloatToStr(c);
end;
	Anweisungsteil der Prozedur
	
	Algorithmus zur Ereignisbehandlung
	hier: Zugriff auf lokale Variablen

end.	**Schlusszeile der Unit** - Ende.

3. Delphi und OOP

3.1. Vorbemerkungen
3.2. Klassen, Instanzen und Vererbung
3.3. Objekteigenschaften, Ereignisse und Methoden

3.1. Vorbemerkungen

Die Komponentenstruktur von Delphi folgt weitgehend dem Konzept der Objektorientierten Programmierung (OOP). Eine umfassende Vermittlung der Begriffswelt, der Strukturen und Denkweisen der OOP würde den Rahmen eines Programmiergrundkurses beim derzeitigen Vorkenntnisstand der Schüler bei weitem sprengen und auch von den Intentionen des gegenwärtigen Lehrplanes wegführen.

Zum Einsteig in die visuelle Programmierung wird zunächst auch kaum das Ziel vorhanden sein, eigene Komponenten zu programmieren; vielmehr wird die zielgerichtete **Nutzung, Kombination und Modifikation vorhandener Komponenten, gepaart mit einfachen Problemlösungsalgorithmen** im Mittelpunkt des Interesses stehen. Einen Großteil der Arbeit mit den Objekten, z.B. Klassendefinition und Instanzendeklaration, erledigt das Programmiersystem ohnehin automatisch im Hintergrund, während der Programmierende im Vordergrund per Mausklick sein Formular zusammenbastelt.

Aus diesem Grunde soll hier lediglich der Versuch stehen, mittels didaktischer Reduktion zu einer eher intuitiven, jedoch ausbaufähigen Sichtweise auf die OOP zu gelangen.

Als Voraussetzung wird an dieser Stelle angenommen, dass die Schüler bereits einige praktische Erfahrungen beim visuellen Programmieren gesammelt haben, Komponenten auswählen, platzieren und bestimmte Eigenschaften sowohl zur Entwurfszeit als auch zur Laufzeit des Programms zielgerichtet ändern können.

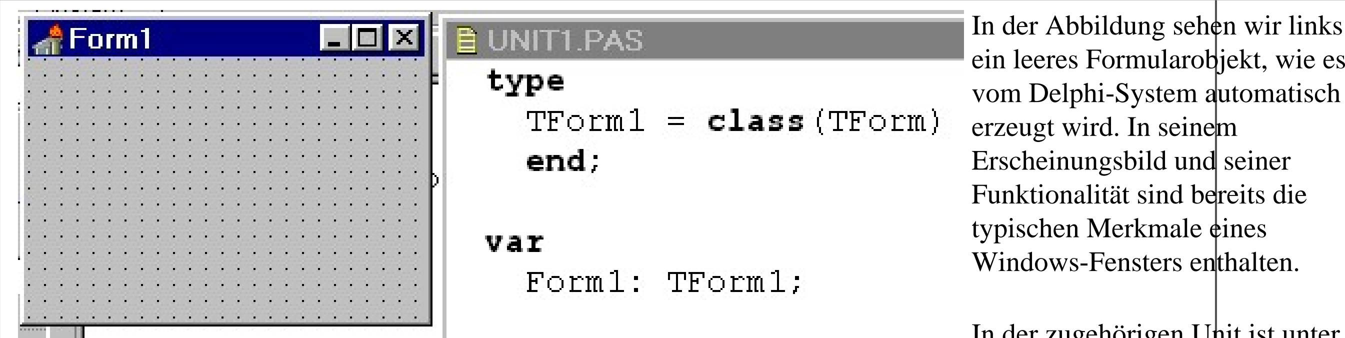
Seitenanfang

3.2. Klassen, Instanzen und Vererbung

a) Begriffserklärungen

Klasse: (Objekttyp)	Aus einer Menge gleichartiger Objekte lassen sich deren typische Merkmale ableiten und daraus Vorschriften für die Bildung neuer Objekte formulieren. Eine solche Vorschrift wird in der OOP als Definition einer Klasse (eines Objekttyps) bezeichnet. Hierbei werden Eigenschaften des Objekts (Erscheinungsbild, Datenfelder) sowie Methoden (Aktionen, die das Objekt ausführen kann) beschrieben.
Instanz: (Objekt)	... ist die auf Grundlage einer Klassendefinition erzeugte Struktur. Im Programm wird eine Instanz (ein Objekt) durch Deklaration einer Variablen eines bestimmten Objekttyps (einer Klasse) erzeugt.
Vererbung:	Objekte (Nachkommen) können andere Objekte (Vorfahren) beerben und erhalten dadurch deren Eigenschaften und Methoden. Die Eigenschaften und Methoden können dabei ergänzt und/oder verändert werden.

b) Verdeutlichung durch Beispiele

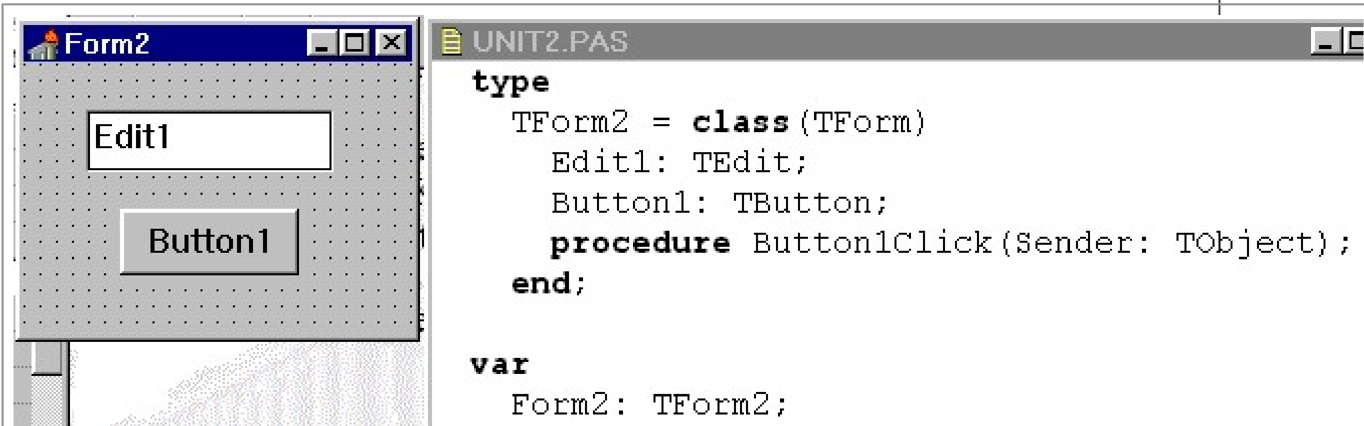


In der Abbildung sehen wir links ein leeres Formularobjekt, wie es vom Delphi-System automatisch erzeugt wird. In seinem Erscheinungsbild und seiner Funktionalität sind bereits die typischen Merkmale eines Windows-Fensters enthalten.

In der zugehörigen Unit ist unter type die Klasse TForm1 definiert, welche alle Merkmale ihres Vorfahren vom Typ TForm erbt.

Unter var wird anschließend ein Objekt (eine Instanz) des Objekttyps (der Klasse) TForm1 deklariert.

Das somit erzeugte Objekt verfügt jedoch gegenüber seinem Vorfahren über keinerlei neue Eigenschaften oder Methoden.



Aus der Komponentenpalette wurden per Mausklick in das Formular 2 ein Edit-Feld und ein Button platziert. Außerdem wurde über den Objektinspektor eine Referenz für das Ereignis Button1Click geschaffen - Voraussetzung dafür, dass das Formularobjekt zur Laufzeit des Programms auf das Anklicken des Buttons reagieren kann.

Die entsprechende Klassendefinition des Objekttyps sagt aus, dass TForm2 alle Merkmale von TForm (Windows-Fenster) erbt und diese durch Einbindung der Komponenten Edit1 und Button1 ergänzt werden. (Dabei erbt Edit1 wiederum alle Eigenschaften des vordefinierten Objekttyps TEdit und Button1 diejenigen von TButton.)
Als neue Methode enthält TForm2 die Ereignisbehandlungsroutine Button1Click, deren Anweisungsfolge weiter unten im Implementationsteil zu programmieren ist.

Schließlich wird wieder unter var ein Objekt (eine Instanz) namens Form2 des Objekttyps (der Klasse) TForm2 deklariert.

Seitenanfang

3.3. Objekteigenschaften, Ereignisse und Methoden

a) Kernaussagen:

1. Delphi folgt mit seinem Konzept weitgehend dem Prinzip der objektorientierten Programmierung, wobei viele **Objekte** in ihrer Grundform bereits als **Komponenten** vorgegeben sind.

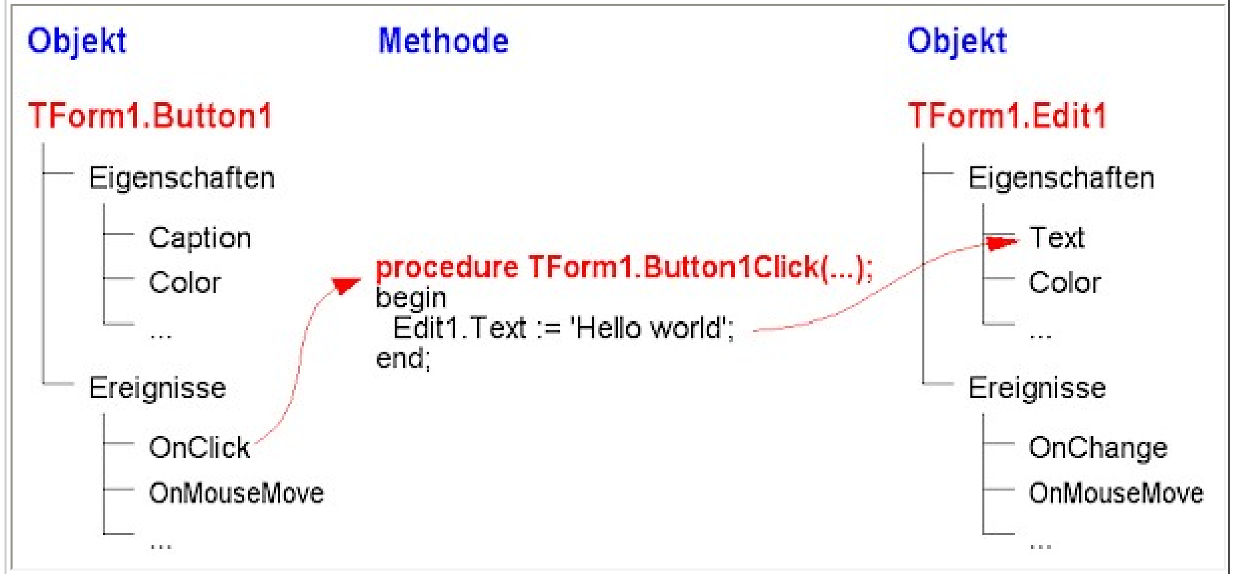
2. **Objekte** (Komponenten) besitzen bestimmte voreingestellte **Eigenschaften**, die zur Entwurfszeit oder zur Laufzeit des Programms geändert werden können.

3. **Methoden** stellen die Fähigkeit von **Objekten** dar, auf bestimmte **Ereignisse** zu reagieren (Ereignisbehandlungsroutinen).

4. Löst zur Laufzeit eines Programms ein **Ereignis** eine **Methode** aus, so verändert diese zielgerichtet bestimmte **Eigenschaften** von **Objekten**.

b) Verdeutlichung an einem Beispiel:

Das Formularobjekt "Form1" enthalte die Komponenten Button1 und Edit1.
Tritt das Button-Click-Ereignis ein, soll die Eigenschaft Text von Edit1 neu belegt werden.



Autor und Quellenangaben

Autor: Mirko Pabst, Informatiklehrer am Lessing-Gymnasium Plauen

Der vorliegenden Dokumentation liegen mehrjährige Unterrichtserfahrungen in etlichen Informatik-Grundkursen der Jahrgangsstufen 11 und 12 zugrunde. Die algorithmischen Inhalte berücksichtigen weitgehend den gegenwärtig gültigen Lehrplan für Informatik Sek. II des Landes Sachsen.

Ich (und hoffentlich auch meine Schüler) habe(n) keine Minute bereut, im Rahmen der imperativen Programmierung von vormals Turbo-Pascal auf das visuelle Programmiersystem Delphi umgestiegen zu sein, obgleich im Anfang manches Problem den Nachtschlaf raubte. ;-)

Interessenten steht diese noch längst nicht vollständige Dokumentation zur freien Nutzung, ggf. Weiterentwicklung zur Verfügung. Rückinformationen über Erweiterungen, Verbesserungen und natürlich auch Kritik nehme ich gerne entgegen.

Plauen, 20.10.1999

Quellen:

Frischalowski / Herdt: ***Borland Delphi 2.0,*** HERDT-Verlag Nackenheim, Germany, 1996

Borland International Inc.: ***Benutzerhandbuch Delphi für Windows,***
Delphi-Online-Hilfe

Fechner, Siegfried: ***Objektorientierte Programmierung,***
Vorlesungsskript Fakultät Informatik der TU Dresden, 1995

Liebrich / Neudecker: ***Delphi Arbeitsmaterialien für den Informatikunterricht,***
CD-ROM, Cornelsen, 1998

Laubach / Knoch: ***Grundkurs Informatik Teil 1 und Teil 2***,
Bayerischer Schulbuchverlag, 1989

Löffler / Meinhardt / Werner: ***Taschenbuch der Informatik***
Fachbuchverlag Leipzig, 1992

Schulze, Hans Herbert: ***Computer Enzyklopädie***
Rowohlt, 1993

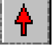
Die nachfolgend vorgestellten und in Erweiterung befindlichen Unterrichtsprojekte bzw. -übungen wurden von mir in der gegebenen Reihenfolge im Unterricht durchgeführt und sollen Beispiele zur Gestaltbarkeit der wesentlichsten Lernbereiche der Programmierung des (gegenwärtigen) sächsischen Lehrplans Informatik Sek. II von 1992 mit dem Programmiersystem Delphi vorstellen.

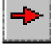
Der zeitliche Umfang der Projekte, die oft verschiedene "Teilprogramme" zu einem Problemkreis über ein Formular realisieren, erstreckt sich meist über mehrere Unterrichtseinheiten, wodurch der Identifikationsgrad der Schüler mit ihrem Softwareprodukt wächst und den realen Gegebenheiten der Programmierertätigkeit näher gekommen wird. Also nicht: eine Stunde - ein Programm, nächste Stunde - nächstes Programm ;-)

Es erfolgt keine explizite Aufgliederung in Tafelbilder, Folien, Arbeitsblätter etc. sondern vielmehr werden kurze Unterrichtsszenarien dargestellt, Hinweise zur Oberflächengestaltung und zum Programmaufbau gegeben sowie, je nach Schwierigkeitsgrad, Orientierungshilfen für den Schüler angeboten.
Dabei wurde meist auf fertige Problemlösungen bzw. die vollständige Wiedergabe von Quelltexten verzichtet, damit dieses Material so wie es ist auch in die Hand des Schülers gegeben werden kann. Gegebenenfalls lassen sich natürlich auch als "zu viel" angesehene Hinweise/Vorgaben bei einer Installation im Intranet löschen.

Quelltextdateien, die zum Download angeboten werden, sind jeweils nur "anprogrammiert", um den Lernenden schnell zum Kern der Aufgabenstellung vordringen zu lassen und nicht die Freude an der eigenen Problemlösung zu nehmen.

Einzelne Programme, die auch über das eigentliche Programmieren hinaus als Unterrichtsmittel zum Einsatz gelangten, wie z.B. zur Effizienzanalyse verschiedener Sortierverfahren, liegen auch als downloadbare EXE-Datei vor.





PÜ 1: Taschenrechner-Programm

Dieses erste Projekt soll den Einstieg in die Delphi-Programmierung erleichtern. Es besteht daher aus nur wenigen Komponenten, die allerdings mehrfach auftauchen und die zu programmierenden Methoden ähneln einander sehr.

Aufgabe:	**Es ist ein nutzerfreundliches Programm zur Realisierung der Grundrechenarten zu erstellen.** Zu einem späteren Zeitpunkt wird das Programm durch weitere Rechenarten wie Potenzieren, Radizieren, Logarithmieren, Fakultätsberechnung usw. ergänzt.
Komponenten:	**Formular, Editierfelder, Buttons, Labels**
Vorschlag zur Oberflächengestaltung:	Die Programmierung der im Formular rot gekennzeichneten Buttons lässt sich nicht durch direkte Eingabe einer Formel realisieren. Da die dazu notwendigen Schleifenstrukturen erst zu einem späteren Zeitpunkt eingeführt werden, bleiben diese Buttons zunächst "unbelegt". Sie können natürlich auch als Anregung zum Weiterprogrammieren für erfahrenere Schüler genutzt werden. ;-)

Quelltext zur Realisierung der Addition:	`procedure TForm1.Button1Click(Sender: TObject);`
	`{Addition}`
	`var a, b, c: Real;`
	`begin`
	`  a := strtofloat(edit1.text);`
	`  b := strtofloat(edit2.text);`
	`  c := a + b;`
	`  edit3.text := floattostr(c);`
	`end;`

Einige ausgewählte Sprachelemente aus Object-Pascal / Delphi:

Sprachelemente	*Erläuterung*	*Beispiel*
	Datentypen	
Real	Typ der reellen Zahlen	**var a, b, c : Real;**
Integer	Typ der ganzen Zahlen	**var x, i: Integer;**

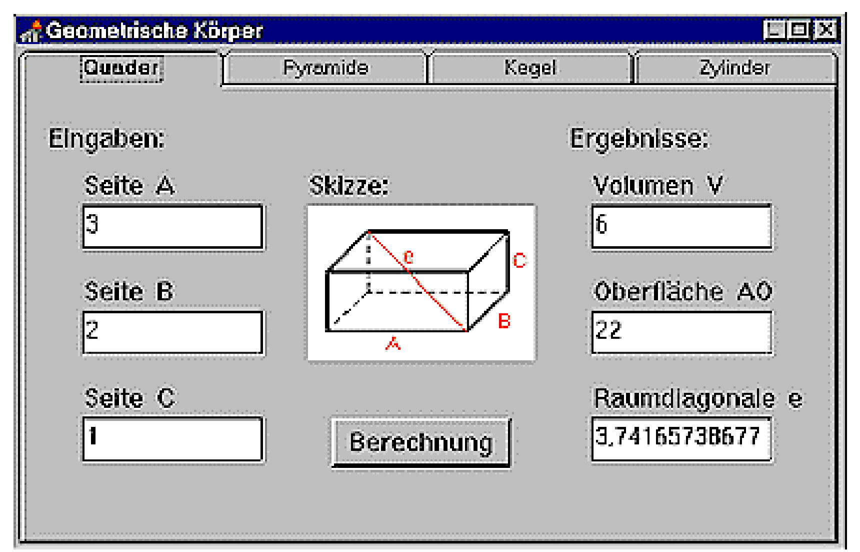
String	Typ einer Zeichenkette	**var s : String;**
	Typumwandelnde Funktionen	
Trunc	Wandelt Real-Werte in Integer-Werte um	**x := trunc(a);**
StrToFloat	Wandelt eine Zeichenkette in eine reelle Zahl um	**a:=StrToFloat(edit1.text);**
FloatToStr	Wandelt eine reelle Zahl in eine Zeichenkette um	**edit3.text:=FloatToStr(c);**
	Arithmetische Funktionen	
Sqrt	Quadratwurzel	**c := Sqrt(a);**
Ln	Natürlicher Logarithmus (Basis e)	**c := ln(b);**
Sin	Sinus	**c := sin(a);**
Cos	Kosinus	**c := cos(a);**
	Methoden	
Close	Formularfenster schließen	**Form1.Close;**

PÜ 2: Berechnung geometrischer Körper

Zielstellung und Szenario:

Vom algorithmischen Anspruch her ordnet sich dieses Projekt in die Thematik "lineare Programmstrukturen" ein.
Als neue Komponenten lernen die Schüler das TabbedNotebook (Karteikasten) sowie das Image (Bilderrahmen zur Aufnahme einer Grafikdatei) kennen.
Da sich die zu programmierenden Seiten des "Karteikastens" äußerlich und vom Quelltext her ähneln, festigen die Schüler ihre Fertigkeiten im Umgang mit den visuellen Komponenten sowie die Umsetzung des E-V-A-Prinzips in linearen Programmstrukturen.

Problemformulierung:



Es ist ein nutzerfreundliches Delphiprogramm zu erstellen, welches für verschiedene geometrische Körper (Quader, Pyramiden, Kegel, Zylinder etc.) die jeweiligen Seitenlängen einliest und auf Knopfdruck die Werte für Oberfläche, Volumen usw. berechnet und ausgibt.

Es soll möglich sein, auf einem einzigen Formular über Registerseiten die jeweils zu berechnenden Körper auszuwählen.

Zur Verbesserung der Übersichtlichkeit ist der betreffende geometrische Körper als Abbildung zu zeigen und seine Abmessungen zu beschriften.

Komponente	*Eigenschaft*	*Wert*	*Bemerkungen*
Form1	*Caption*	‘Geometrische Körper’	Überschrift in der Kopfleiste des Programmfensters
TabbedNotebook1 *(Karteikasten)*	*Align*	‘alClient’	Komponente füllt das ganze Formular aus und wird mit diesem vergrößert bzw. verkleinert

	Pages	'Quader' 'Pyramide' 'Kegel' 'Zylinder'	Die Benennung der einzelnen Registerseiten wird hier festgelegt (TStrings). Jede zu TStrings hinzugefügte Benennung eröffnet automatisch eine neue Seite im Register.
	TabsPerRow	4	So viele Registerseiten erscheinen maximal nebeneinander.
	ActivePage	z.B. 'Quader'	Name der gerade zu programmierenden Seite (Auswahl aus Pages). Diese Seite wird beim späteren Programmstart zur aktuellen Seite.
Image1	***Picture***	quader.bmp	Einfügen einer Grafik in ein Formular.

Quelltext zur Realisierung der Quaderberechnung:

```
procedure TForm1.Button1Click (Sender: TObject);
var a, b, c, v, ao, e: Real;
begin
  {Einlesen der Variablen}
  a := StrToFloat(Edit1.Text);
  b := StrToFloat(Edit2.Text);
  c := StrToFloat(Edit3.Text);
  {Berechnung}
  v := a*b*c;
  ao:= 2*(a*b+a*c*b*c);
  e := Sqrt(Sqr(a)+Sqr(b)+Sqr(c));
  {Ausgabe der Ergebnisse}
  Edit4.Text := FloatToStr(v);
  Edit5.Text := FloatToStr(ao);
  Edit6.Text := FloatToStr(e);
end;
```

PÜ 3: Alternative Strukturen

PÜ 3.1. Übungen zur einseitigen Alternative
- PÜ 3.1.1. Programm zur Rabattberechnung
- PÜ 3.1.2. Berechnung und Auswertung des Kraftstoffverbrauches (I)

PÜ 3.2. Übungen zu zweiseitigen Alternativen und Verbundanweisungen
- PÜ3.2.1. Quadratwurzel mit Prüfung auf negativen Eingabewert
- PÜ3.2.2. Kraftstoffverbrauch Variante II

PÜ3.1. Übungen zur einseitigen Alternative

Grundstruktur:

```
IF Bedingung THEN Anweisung;
```

Die Anweisung hinter THEN wird nur dann ausgeführt, wenn die vorgegebene Bedingung erfüllt ist.

PÜ3.1.1. Programm zur Rabattberechnung

a)	**Gegeben** sind die Real-Variablen *anzahl* und *einzelpreis*; **Zu berechnen** sind *grundpreis*, *rabattsatz*, *rabatt*, und *endpreis*, wenn folgendes gilt: Grundsätzlich wird kein Rabatt gewährt. Falls die Anzahl gleich oder größer 100 beträgt, wird ein Rabattsatz von 5% angerechnet. Notieren Sie die nötigen Programmzeilen!
b)	Entwerfen Sie ein Formular gemäß der nebenstehenden Abbildung und fügen Sie den unter 1. erarbeiteten Programmtext zur Ereignisbehandlung von ***Button1Click*** (Kasse) ein! Beachten Sie, dass als Eingabevariablen nur *anzahl* (über Edit1.Text) und *einzelpreis* (über Edit2.Text) einzulesen sind. Alle übrigen Textfelder dienen nur der Ausgabe! Testen Sie mehrfach unter Berücksichtigung der bedingten Rabattvergabe! Sichern Sie das Projekt als *rabatt1.dpr*! Rechnung mit Rabatt Stückzahl: 450 Einzelpreis: 22,80 DM — Kasse Grundpreis: 10260 DM Rabattsatz: 5 % Rabatt: 513 DM — Löschen Endpreis: 9747DM — Ende

c)	Das Programm ist unter **Verwendung einseitiger Alternativen** mit folgenden gestaffelten Rabattvergaben zu erweitern:

Anzahl	Rabattsatz
`0...   99`	`0%`
`100...  499`	`5%`
`500...  999`	`10%`
`1000...1499`	`15%`
`1500...1999`	`20%`
`ab 2000`	`30%`

Sichern Sie nun das erweiterte Projekt unter gleichem Namen !

PÜ3.1.2. Berechnung und Auswertung des Kraftstoffverbrauches

Es ist ein Programm zu entwickeln, welches nach Eingabe der gefahrenen Kilometer und verbrauchten Liter Kraftstoff den Kraftstoffverbrauch pro 100 km ermittelt und ausgibt.

Durch bedingte Anweisungen soll das Programm wie folgt reagieren:
- falls der Verbrauch > 8 l/100km, dann Ausgabe: 'Verbrauch ist zu hoch!'
- falls der Verbrauch < 4 l/100km, dann Ausgabe: 'Verbrauch ist unrealistisch!'

a)	Entwerfen Sie unter Delphi ein geeignetes Formular!
b)	Programmieren Sie eine ButtonClick-Ereignisbehandlung, welche obiger Aufgabe entspricht!
c)	Test / Korrektur / Sicherung

Seitenanfang

PÜ3.2. Übungen zu zweiseitigen Alternativen und Verbundanweisungen

Grundstruktur der zweiseitigen Alternative:

```
IF Bedingung THEN Anweisung_1 ELSE Anweisung_2;
```

Die Anweisung hinter THEN wird ausgeführt, wenn die vorgegebene Bedingung erfüllt ist, andernfalls wird die Anweisung hinter ELSE ausgeführt.

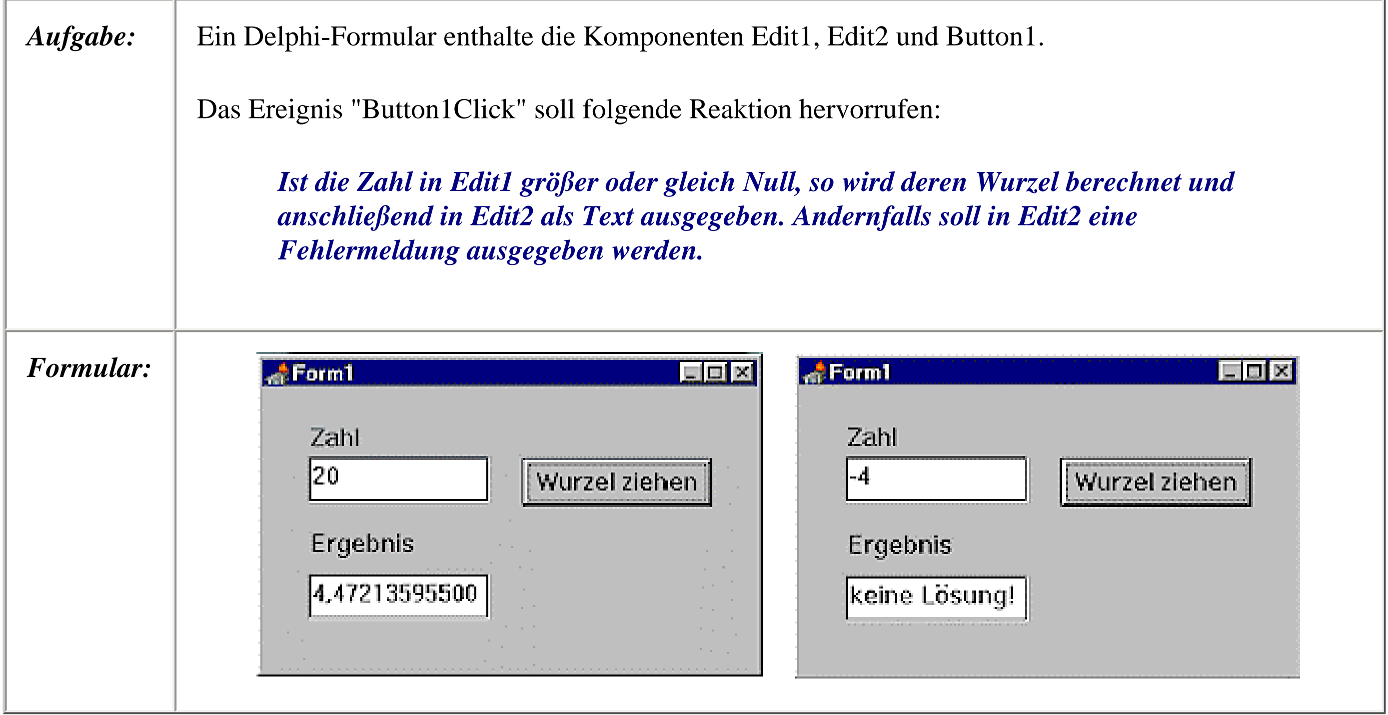
Grundstruktur der Verbundanweisung:

```
IF Bedingung THEN
  BEGIN
    Anweisung_1;
    Anweisung_2;
    ...
    Anweisung_n
  END
ELSE
   ...;
```

Soll in einer syntaktischen Struktur eine Folge von n Anweisungen genau so behandelt werden, als würde es sich nur um eine einzige Anweisung handeln, so verwende man `begin` und `end` im Sinne von mathematischen Klammern.

Zur Bewahrung der Übersicht in zunehmend komplexer werdenden Quelltexten ist es dringend anzuraten, sich unterordnende bzw. eingeschachtelte Teilstrukturen nach rechts einzurücken.
Im Beispiel ordnet sich das "BEGIN" dem "IF-THEN" unter und die "Anweisung_1" wiederum dem "BEGIN" usw.

PÜ3.2.1. Quadratwurzel mit Prüfung auf negativen Eingabewert



Aufgabe:	Ein Delphi-Formular enthalte die Komponenten Edit1, Edit2 und Button1. Das Ereignis "Button1Click" soll folgende Reaktion hervorrufen: ***Ist die Zahl in Edit1 größer oder gleich Null, so wird deren Wurzel berechnet und anschließend in Edit2 als Text ausgegeben. Andernfalls soll in Edit2 eine Fehlermeldung ausgegeben werden.***
Formular:	

PÜ3.2.2. Kraftstoffverbrauch Variante II

Das Programm zur Berechnung und Auswertung des Kraftstoffverbrauches ist folgendermaßen zu erweitern:

1. Falls der Verbrauch im normalen Bereich liegt, also ***4 l/100km <= verbrauch <= 8 l/100km*** soll die Ausschrift erfolgen: ***"normaler Verbrauch"***.
2. Zur optischen Unterstützung der Ausgabe soll das entsprechende Edit-Feld bei normalen Verbrauch grün, bei zu hohem Verbrauch rot und bei unrealistisch geringem Verbrauch gelb eingefärbt werden.

Unter Verwendung der zweiseitigen Alternative und der Verbundanweisung entwerfe man zunächst ein Struktogramm und erweitere danach das bereits vorhandenen "Benzinprogramm".

Ideen zur Erweiterung:

Die vorliegenden Gegebenheiten entsprechen einem "Allerweltsauto" mit 1300cm³ - Benzinmotor und 60 PS, decken also nur eine stark begrenzte Problemklasse ab.

- Ermitteln Sie Verbrauchsnormative für Pkws mit stärkerer und schwächerer Motorisierung, getrennt nach Benzin- und Dieselmotor.
- Modifizieren Sie danach die Sollwerte für Minimal- und Maximalverbrauch (bisher konstant 4 bzw. 8 l/100km) als Variablen mit entsprechenden prozentualen Ab- oder Zuschlägen für Motorart, Leistung und Hubraum.
- Nach Erweiterung der Oberfläche um die zusätzlichen Eingabekomponenten ist der Algorithmus entsprechend zu verfeinern und in das Programm zu implementieren.

Zielstellung und Szenario:

Dieses Programmierobjekt soll das Zusammenspiel von Objekteigenschaften, Ereignissen und Methoden verdeutlichen, indem an einem Edit-Feld über Button-Click-Ereignisse bestimmte Eigenschaften zur Laufzeit per Wertzuweisung geändert werden.

Zur Zeiteinsparung bekommen die Schüler das unten abgebildete Formular bereits als Datei vorgegeben.

Der Lehrende demonstriert anhand der Eigenschaft Edit1.Visible die Benutzung der Online-Hilfe von Delphi und setzt die daraus gewonnenen Informationen in die entsprechenden Prozeduren um.
Die Schüler nutzen anschließend diese Vorgehensweise und vervollständigen systematisch die noch fehlenden Prozeduren für die übrigen Buttons. Zur Ergebnissicherung notieren sie ihre Erkenntnisse in einer Tabelle.

Formular:

Zusammen-fassung:

Eigenschaft	Bedeutung	Beispiel für eine Wertzuweisung
Visible	Sichtbarkeit der Komponente	`Edit1.Visible := False;`
Enabled	steuert, ob die Komponente auf Maus-, Tastatur- und Timerereignisse reagiert	`Edit1.Enabled := True;`
Color	hiermit wird die Hintergrundfarbe der Komponente festgelegt.	`Edit1.Color := clRed;`
Width	bestimmt die horizontale Größe des Dialogelements oder Formulars	`Edit1.Width := Edit1.Widht+10;`
Height	bestimmt die vertikale Größe des Dialogelementes oder Formulars	`Edit1.Height := Edit1.Height-10;`
Font.Color	bestimmt die Schriftfarbe innerhalb des Dialogelementes	`Edit1.Font.Color := clGreen;`

Beispiel-prozeduren:	see code below

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Form1.Caption := Edit1.Text;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Edit1.Visible := False;
end;
```

PÜ 5: Prämienberechnung

Zielstellung und Szenario:

In den vergangenen Jahren mutierte die Struktogrammdarstellung oft zum Fetisch, indem die Schüler angehalten wurden, doch jede auch noch so profane Problemlösung zunächst als Struktogramm "aufzumalen".
Deren Gegenreaktion: Oft wurde der Algorithmus gleich in Programmiersprache ausformuliert und erst im Nachgang ein liebloses Struktogramm dazu hingezeichnet - eine den Algorithmiker schmerzende Vorgehensweise.

Das nachfolgende Problem ist ausreichend einfach für den Unterricht, aber kompliziert genug, um die meisten "Sofortprogrammierer" an ihre Grenzen zu führen. Ergo soll es die Zweckmäßigkeit von Struktogrammen bewusst machen und gleichzeitig den sicheren Umgang mit ihnen festigen.

Problemformulierung:

Unter folgenden Bedingungen zahlt ein Betrieb an seine Mitarbeiter eine Jahresendprämie:
Beträgt das Betriebszugehörigkeit weniger als ein Jahr, wird keine Prämie gezahlt;
beträgt sie mindestens ein Jahr aber weniger als 6 Jahre, erhält man 200 DM.
Bei einer Betriebszugehörigkeit von sechs oder mehr Jahren bekommt man 80 DM dazu und für jedes geleistete Dienstjahr 20 DM.
Beträgt im letzteren Falle das Lebensalter 50 oder mehr Jahre, so wird nochmals eine Zulage von DM 50 gezahlt.

Problemanalyse (Vorbetrachtung)

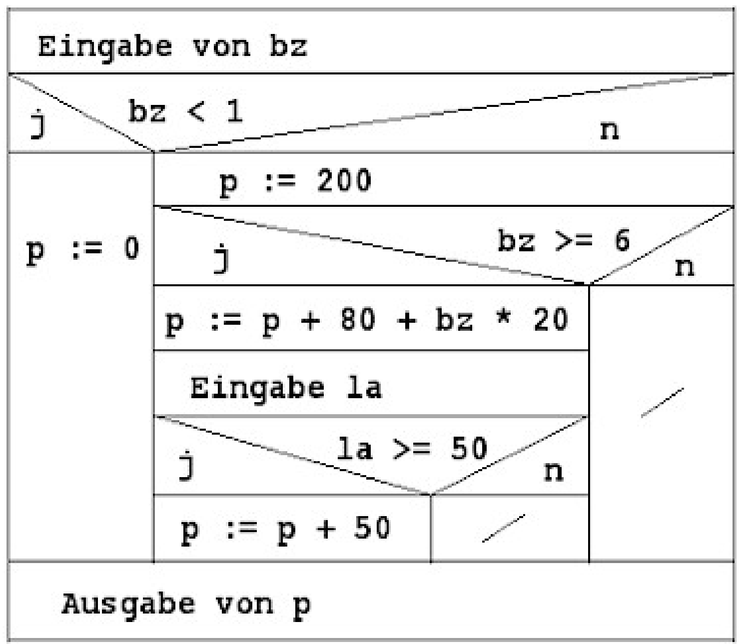
Betriebszu-gehörigkeit	Lebensalter	Prämie
3 Jahre	28	?
7 Jahre	38	?
12 Jahre	48	?
20 Jahre	58	?

Zunächst wird diese Tabelle angeschrieben, und nach vermutlichen Lösungen für die einzelnen Fälle gefragt. Mit ziemlicher Sicherheit kommen für jeden Fall verschiedene Antworten zustande, die mit Fragezeichen versehen und notiert werden.
Nach Entwicklung des Struktogramms werden die Antworten verifiziert und nach der Programmerstellung noch einmal.
Der Effekt ist meist verblüffend ... ;-)

Lösungsalgorithmus als Struktogramm:

Variablen:
bez.: Betriebszugehörigkeit,
la: Lebensalter,
p: Prämie.

Anmerkung:
Zur inhaltlich korrekten Darstellung der obigen Formulierungen in Struktogrammform sind mehrere (vielleicht weitaus elegantere) Lösungen denkbar.
Im vorliegenden Fall wurde bewusst eine abfolgegleiche grafische Repräsentation des



verbalen Algorithmus gewählt, um den Einsteiger die inhaltliche Übereinstimmung beider Darstellungsformen besser sichtbar zu machen.

Formular:

Um sukzessive mit weiteren Komponenten vertraut zu machen, wurden hier zwei RadioButtons, angeordnet in einer GroupBox, zur "Eingabe" des Lebensalters gewählt.
Da der Algorithmus nicht das genaue Alter benötigt, scheint diese Variante zur Erhöhung der Nutzerfreundlichkeit recht sinnvoll, reicht doch nunmehr ein Mausklick zur Eingabe der Altersstufe aus.
Bei der Programmierung der Ereignisbehandlungsroutine müssen diese neuen Eingabekomponenten natürlich berücksichtigt werden.

Prozedur zur Berechnung

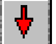
```
procedure TForm1.Button1Click(Sender: TObject);
  var bez., p: integer;
  begin
    bez. := StrToInt(Edit1.Text);
    if bez. < 1 then
      p := 0
    else begin
      p := 200;
      if bez. >= 6 then begin
        p := p + 80 + bez. * 20;
        {Falls Lebensalter >= 50}
        if RadioButton2.Checked then
          p := p + 50;
      end;{of if}
    end;{of else}
    Edit2.Text := IntToStr(p);
  end;
```

Inhalt

PÜ6: Dreiecksanalyse

Downloads: dreieck1.exe (fertiges Programm)
dreieck1.zip (vorgefertigtes Projekt mit Grafik-Funktion)

Aufgabenstellung
Algorithmisch-programmiertechnische Grundlagen
- Struktogramm der Abfragesituation
- Verknüpfung von Bedingungen durch boolesche Operatoren (AND / OR)
- Vorschlag zur Oberflächengestaltung
- Verwendung von TListBox-Komponenten
- Verwendung von TPaintBox-Komponenten



Aufgabenstellung:

Die Seitenlängen eines Dreieckes sind einzulesen. Stellen Sie fest,

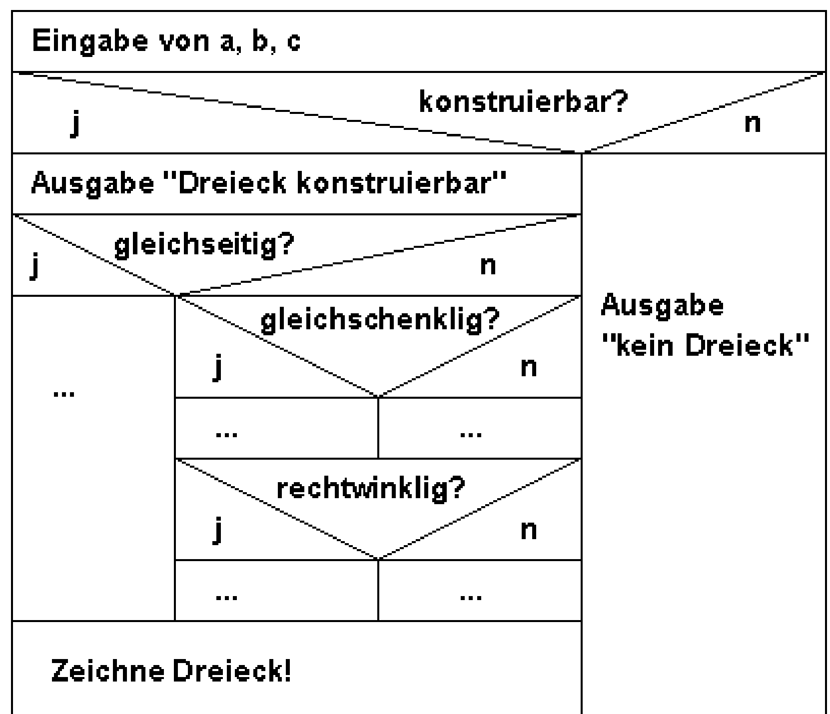
- ob mit diesen Angaben ein Dreieck konstruiert werden kann und wenn ja,
- ob es sich um ein gleichseitiges, ein gleichschenkliges oder ein ungleichschenkliges Dreieck handelt und
- ob das Dreieck einen rechten Winkel besitzt.

a)	Gegeben ist ein Struktogramm in Grobform. Formulieren Sie die abzufragenden Bedingungen verbal bzw. durch mathematische Aussagen aus! *Beispiel:* Konstruierbarkeit
b)	In den Abfragen sind offensichtlich mehrere Einzelbedingungen miteinander zu verknüpfen. Machen Sie sich mit der Verwendung der Boolschen Operatoren AND und OR vertraut!
c)	Erstellen Sie unter Delphi eine geeignete Oberfläche für Ihr Programm! (Vorschlag) Da die Ausgabe des "Analyseergebnisses" mehrzeilig ausfällt, sind die bisher bewährten Edit-Felder hierzu nicht geeignet - es empfiehlt sich die Verwendung einer ListBox-Komponente.

Seitenanfang

Algorithmisch-programmiertechnische Grundlagen:

Struktogramm der Abfragesituation



Konstruierbarkeit eines Dreiecks:

Ein Dreieck mit den Seiten a, b und c ist dann konstruierbar, wenn gilt:
(a>b+c) und (b>a+c) und (c>a+b)

Anmerkung: Die obige Vorgehensweise widerspiegelt eine typisch informatische Problemlösestrategie, die sog. Top-Down-Methode (Methode der schrittweisen Verfeinerung), nach der ein komplexer Sachverhalt schrittweise in überschaubare Teilkomplexe zerlegt wird.
Mit anderen Worten: Es wäre verwirrend, wollte man gleich bei der Struktogrammerstellung die konkreten Abfragebedingungen für Konstruierbarkeit usw. fertig ausformulieren.

Seitenanfang

Verknüpfung mehrerer Bedingungen in Auswahlstrukturen durch Boolesche Operatoren

Oft ist die Fortsetzung eines Programms von mehreren Bedingungen abhängig, von denen entweder nur eine oder einige oder aber alle gleichzeitig erfüllt sein müssen.

Verknüpfung mit AND:

```
IF (Beding1) AND (Beding2) AND (Beding n) THEN Anweisung;
```

- Anweisung wird nur dann ausgeführt, wenn alle aufgeführten Bedingungen erfüllt sind!

Verknüpfung mit OR:

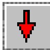
```
IF (Beding1) OR (Beding2) OR (Beding n) THEN Anweisung;
```

- Anweisung wird immer dann ausgeführt, wenn mindestens eine der Bedingungen erfüllt ist!

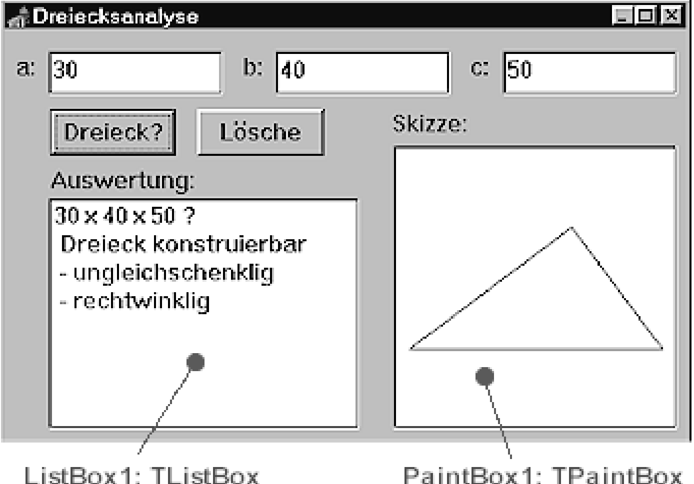
Beispiel: Von drei Zahlen (a, b, c) ist das Maximum zu ermitteln.

```
IF (a>b) AND (a>c) THEN max := a
ELSE
  IF (b>a) AND (b>c) THEN max := b
  ELSE
    IF (c>a) AND (c>b) THEN max := c
    ELSE ShowMessage ('Die Zahlen sind nicht verschieden !');
```

Seitenanfang



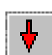
Vorschlag zur Oberflächengestaltung:



Zur "optischen Aufwertung" der Programmoberfläche empfiehlt sich der Einbau einer Grafikkomponente (PaintBox), in welcher für den Fall der Konstruierbarkeit das Dreieck zur Anzeige gelangt. Hierbei sollte eine Art Zoom-Funktion einprogrammiert werden, so dass ein Dreieck der Größe 3; 4; 5 ebenso groß erscheint wie eines der Größe 3000; 4000; 5000.

Methodische Varianten:
a) die entsprechende Prozedur wird von erfahreneren Schülern entwickelt (Hilfestellung) und dann für alle verfügbar gemacht oder
b) es wird vom Lehrer eine vorgefertigtes Projekt zur Verfügung gestellt, in dem bereits die zum Zeichnen notwendigen Komponenten nebst Prozedur enthalten sind.
(Download von dreieck1.zip)

Seitenanfang

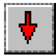


Verwendung von TListBox-Komponenten:

Komponenten vom Typ TListBox werden auch als Windows-Listenfenster bezeichnet. Als zentrale Eigenschaft steht "Items" zur Verfügung, welche die Einträge der Liste vom Typ String enthält.
Da wir uns in einer folgenden Aufgabenstellung noch ausführlich mit TListBox-Komponenten beschäftigen, erfolgen hier nur zwei elementare Hinweise zur unmittelbaren Nutzung im Programmbeispiel:

- **Anfügen eines Texteintrages an die Stringliste in ListBox1:**
```
ListBox1.Items.Add ('Beliebiger Text');
```
- **Löschen aller Texteinträge in ListBox1:**
```
ListBox1.Clear;
```

Seitenanfang



Verwendung von TPaintBox-Komponenten:

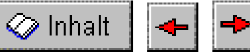
Die Komponente TPaintBox bietet eine Möglichkeit, in einem definierten rechteckigen Bereich des Formulars zu zeichnen, dies jedoch außerhalb des Zeichenfelds zu unterbinden - quasi eine definierte Zeichenfläche im Formular. Das Zeichenfeld enthält Koordinaten, wobei der Punkt P1(**0,0**) die **linke obere Ecke** und P2(PaintBox1.Height, PaintBox1.Width) die rechte untere Ecke darstellt.
Hinweise zur Verwendung:

- **"Löschen" der PaintBox:**
  Dies kann durch "übermalen" des aktuellen Inhaltes mit einem weiß ausgefülltem Rechteck erfolgen, dessen Größe der Paintbox entspricht.
  `PaintBox1.Canvas.Rectangle (0,0,paintbox1.height,paintbox1.width);`
- **Zeichnen einer Linie:**
  `PaintBox1.Canvas.LineTo(x,y);`
  Gezeichnet wird von der gegenwärtigen Stiftposition (PenPos) zu dem durch x/y beschriebenen Punkt.
- **Verändern der Stiftposition:**
  `PaintBox1.Canvas.MoveTo(x,y);`
  Hierbei wird der Stift (ohne dass er eine Linie zeichnet) zur Position x/y bewegt.

Beispiel: In eine quadratische PaintBox soll ein gleichseitiges Dreieck gezeichnet werden, dessen Spitze nach oben zeigt

```
procedure Gleichseitiges_Dreieck;
  var a,b,c: TPoint;
      seitenlaenge: Integer;
  begin
    with PaintBox1 do begin
      {Bestimmen der Koordinaten für a,b,c}
        a.x := 1;
        a.y := height-1;
        b.x := width-1;
        b.y := a.y;
        seitenlaenge := b.x-a.x;
        c.x := a.x + seitenlaenge DIV 2;
        c.y := a.y - trunc(sqrt(3/4*sqr(seitenlaenge)));
      {Zeichnen durch Verbinden der Eckpunkte}
        canvas.moveto(a.x, a.y);
        canvas.lineto(b.x, b.y);
        canvas.lineto(c.x, c.y);
        canvas.lineto(a.x, a.y);
    end;{of with}
end;
```

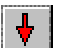
PÜ 7: Arbeit mit TListBox-Komponenten



Zielstellung und Szenario:

TListBox-Komponenten stellen für einfache Programme ein geeignetes Mittel dar, um die Ausgabe größerer Datenmengen zu visualisieren. Insbesondere bei den nachfolgenden Übungen, die sich mit Schleifen beschäftigen, sollte der sichere Umgang mit Listboxen zum Repertoire der Schüler gehören.
Darüber hinaus verfolgt die vorliegende Übung das Ziel, die effektive Nutzung der interaktiven Hilfe-Funktionen von Delphi weiter zu trainieren.

Seitenanfang



Aufgabenstellung:

Die Komponente TListBox ist ein Listenfeld in Windows. In einem Listenfeld wird eine Liste von Strings (Zeichenketten) angezeigt, aus der ein oder mehrere Listenelemente ausgewählt werden können

---

1. Erstellen Sie unter Delphi ein Formular gemäß der nachfolgenden Vorgabe!

2. Informieren Sie Sich in der Delphi-Hilfe über die wichtigsten Eigenschaften und Methoden der Komponente TListBox und realisieren Sie danach folgende OnClick-Ereignisbehandlungen:

a) **Button1 (Hinzu)**
Der Text von Edit1 soll an die Liste in ListBox1 angefügt werden. Anschließend ist der Inhalt von Edit1 zu löschen und die Eigenschaft Form1.ActiveControl auf Edit1 zu setzen.
(Eigenschaft Items, Methode Add)

b) **Button2 (Hinweg)**
Der gerade markierte Listeneintrag von ListBox1 (z.B. Hans) soll aus der Liste entfernt werden.
(Eigenschaft Items, Methode Delete sowie Eigenschaft Itemindex)

c) **Button3 (Sortiere)**
Die ListBox soll in sortierter Form erscheinen und alle folgenden Einträge sollen in die Sortierung eingefügt werden.
(Eigenschaft Sorted)

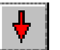
d) **Button4 (Lösche alles)**
Die gesamte ListBox soll gelöscht und die Sortierung aufgehoben werden.
(Methode Clear, Eigenschaft Sorted)

3. Speichern Sie das Projekt unter "Listbox1.dpr" und drucken Sie Sich die Prozeduren zur Ereignisbehandlung aus!

Zusatzaufgabe:

Fügen Sie im fertiggestellten Programm die Zahlen von 1 bis 20 in umgekehrter Reihenfolge in das Listenfeld ein und betätigen Sie anschließend den "Sortiere-Button"!
Achten Sie auf die sich ergebende Sortier-Reihenfolge und begründen Sie deren Zustandekommen!
Finden Sie Möglichkeiten, die Zahlen trotzdem chronologisch zu sortieren?!

Seitenanfang



Auszug aus der Delphi-Hilfe

Komponente TListBox

Unit StdCtrls

Beschreibung

Die Komponente TListBox ist ein Listenfeld in Windows. In einem Listenfeld wird eine Liste angezeigt, aus der ein oder mehrere Listenelemente ausgewählt werden können.
Diese Liste ist der Wert der Eigenschaft **Items**. Die Eigenschaft **ItemIndex** zeigt an, welches Listenelement gerade ausgewählt wurde.
Mit den Methoden **Add**, **Delete** und **Insert** des Objekts Items, das vom Typ TStrings ist, lassen sich Listenelemente anfügen, löschen und einfügen. So würde man zum Beispiel einen String in einem Listenfeld mit folgender Programmzeile anfügen: `ListBox1.Items.Add('Neues Element');`

Auch das Erscheinungsbild des Listenfelds ist änderbar. So kann man ein mehrspaltiges Listenfeld durch Änderung des Wertes der Eigenschaft Columns erzeugen. Die Eigenschaft **Sorted** ermöglicht eine Sortierung der Listenelemente.
...

Einführung zyklischer Strukturen

- Quadratwurzel nach Heron -

Vorbemerkungen
Der Iterationsbegriff
Einführung des Nichtabweisenden Zyklus -Quadratwurzel nach Heron-
Szenario
Der Heronsche Algorithmus - an einem Zahlenbeispiel vorgestellt
Umsetzung mit einer Nichtabweisenden Schleife
Programmierung der Quadratwurzel nach Heron
a) Grundvariante
b) Erweiterte Variante
- *Eingabesicherung mittels Repeat-Schleife*
- *Formatierung mittels FloatToStrF*
- *Zählfunktion in Schleifen*
- *Eingabefunktion mittels TScrollBar*
c) Zusatzaufgabe - Kubikwurzel nach Heron

Vorbemerkungen:

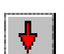
Dem Pascal-Programmierer mag es nicht schwer fallen, die Arbeit mit zyklischen Strukturen zu motivieren: man schreibe ein überschaubares lineares Programm und stelle dann die Frage, wie dem armen Nutzer zu helfen sei, der mit diesem Programm Hunderte von Berechnungen durchführen soll.
Die Lösung liegt schnell auf der Hand: man packe das ganze in eine Repeat-Schleife an deren Ende eine Nutzerauswahl "Neue Berechnung?" / "Ende?" erfolgt und entbinde damit den potentiellen Nutzer von der lästigen Aufgabe, für jede neue Berechnung auch einen neuen Programmstart veranlassen zu müssen.

Unter Delphi greift diese Motivierung natürlich nicht, denn bis zum ultimativen "Form1.Close", verbunden mit einem "Ende"-Button, kann ein Nutzer die Funktionalität des Programm-Fensters so oft strapazieren, wie er möchte, da dank der Ereignisorientierung ohnehin ein zyklisches Abfragen eingabesensitiver Komponenten erfolgt.
Bewährt hat sich m. E. der **Einstieg über einen einfachen zyklischen Algorithmus**, etwa der Iteration einer Quadratwurzel nach Heron. Hieraus werden für das Verständnis von Schleifen notwendige Erkenntnisse, wie z. B. die Bedeutung der Abbruchbedingung abgeleitet und dann auf weitere Beispiele angewandt.

Nichtabweisender Zyklus / Abweisender Zyklus / Zählzyklus ?
Immer häufiger hört man die Frage *"Müssen die Schüler denn wirklich (noch) alle drei Schleifenarten beherrschen?"* Mein Standpunkt: wenn die Zeit dazu vorhanden ist, so hat 's noch keinem geschadet. Jedoch fehlt es meist an eben dieser Zeit. Nach dem Motto **"Weniger ist meistens mehr"** würde ich es für sinnvoller halten, wenn der Schüler im Extremfall mit nur einer einzigen Schleifenart eine Vielzahl von Problemen aufbereiten und lösen kann, als dass er zwar alle Schleifenarten formal kennt, jedoch kaum in der Lage ist, diese selbständig zur Problemlösung zu verwenden.

Die Kunst des Weglassens sollte unbedingt die Überlegung einschließen, dass "Repeat-" und "While-Schleifen" (also Nichtabweisende und Abweisende Zyklen) gegenüber dem Zählzyklus die universelleren Werkzeuge darstellen, d. h. alle auf iterativem Wege lösbaren Probleme lassen sich damit umsetzen, während mit der Zählschleife nur eine Teilmenge dieser Lösungen (sauber) programmiert werden kann.
In meinen Kursen empfand ich es bisher als gangbaren Kompromiss, das **Wesen zyklischer Algorithmen** zunächst anhand der Repeat-Schleife zu verdeutlichen und nachfolgend "so zu tun", als gäbe es nur diese eine Schleifenart. Mit dieser Strukturkenntnis werden alsdann bei zunehmender Selbständigkeit einfache Problemstellungen aus verschiedenen Gebieten bearbeitet und erst am Ende werden eher überblicksartig die beiden verbleibenden Schleifenarten vorgestellt und Vergleiche dazu angestellt.

Seitenanfang

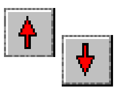


Der Iterationsbegriff:

> ***"Aus dem lat. iteratio = Wiederholung. Verfahren der numerischen Mathematik, das dazu dient, durch wiederholtes Durchlaufen eines bestimmten Rechenverfahrens die Berechnung eines Wertes mit immer größerer Genauigkeit zu erreichen...***
> ***Im übertragenen Sinne verwendet man den Ausdruck I. auch für die Wiederholung eines Schleifendurchlaufes in einem Programm.***
> ***Wesentlich für jede Iteration ist, dass das Ergebnis des ersten Schrittes als Eingabewert für den folgenden Schritt verwendet wird.***
> ***Die Iteration wird im Allgemeinen durch die Zahl der Durchläufe beendet oder durch eine Bedingung, deren Erreichen ebenfalls zum Abbruch der Iteration führt [Abbruchbedingung] ..."***
>
> *Computer-Enzyklopädie S. 1563*

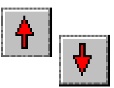
Seitenanfang

Einführung des Nichtabweisenden Zyklus - Quadratwurzel nach Heron -



Szenario:

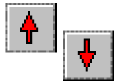
Interessant ist es Schüler der heutigen Generation mit der Frage zu konfrontieren, wie denn wohl unsere Vorfahren ohne Hilfsmittel die Quadratwurzel einer natürlichen Zahl "gezogen" haben. Der Gedanke, dass dies durch Näherungsverfahren geschehen sein muss, ist schnell geäußert; die Frage aber, wie man denn mit vertretbarem Aufwand eine möglichst hohe Genauigkeit erreichen konnte, bleibt meist unbeantwortet. Ein knapp 2000 Jahre altes Verfahren, entwickelt durch ***HERON von Alexandria*** (ca. 20-62 n. Chr.), löst das Problem auf geniale und aus heutiger Sicht sehr computerfreundliche Weise!



Der Heronsche Algorithmus - an einem Zahlenbeispiel vorgestellt:

Wir stellen uns vor, die Quadratwurzel der Zahl 2 sei gesucht. Von der zu findenden Lösung wissen wir bereits, dass deren Quadrat wiederum 2 ergeben muss.
Als Ausgangspunkt wählen wir ein Rechteck, dessen Seitenlänge a = 1 ist; Seitenlänge b ist so groß wie die zu radizierende Zahl, also 2.   Der sich ergebende Flächeninhalt gleicht somit dem Quadrat der gesuchten Zahl.
Über das arithmetische Mittel aus a und b wird jetzt b neu bestimmt, also b' = (a+b) / 2 = 1,5.
Unter Beibehaltung des Flächeninhaltes A=2 ermitteln wir jetzt den neuen Wert für a, also a' = A / b' = 2 / 1,5 = 1,333...
Das Verfahren wird so oft wiederholt, bis eine gewünschte Genauigkeit erreicht ist, d. h. die Differenz aus b und a ausreichend gering ist.
Nach unendlich vielen Iterationsschritten wäre a = b = Quadratwurzel der Zahl 2.

Ausgangspunkt:	***1. Iterationsschritt:***	***... nach unendlich vielen Iterationsschritten:***
b = 2 A = 2 a = 1	b = 1,5 A = 2 a = 1,333... b' = (a+b)/2 a' = A / b'	$b = \sqrt{2}$ A = 2 $a = \sqrt{2}$



Umsetzung des Heronschen Algorithmus mit einer Nichtabweisenden Schleife:

Struktogramm:

Deklariere	zahl: ganzzahlig, a, b, fehler: reell.
Eingabe von zahl und fehler	
a := 1	b := zahl
Wdh.	b := (a+b) / 2 a := zahl / b
bis b - a < fehler	
Ausgabe von b	

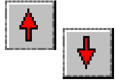
Quelltext:

```
...

a:=1; b:=zahl;
repeat
  b:=(a+b)/2;
  a:=zahl/b;
until b-a < fehler;

...
```

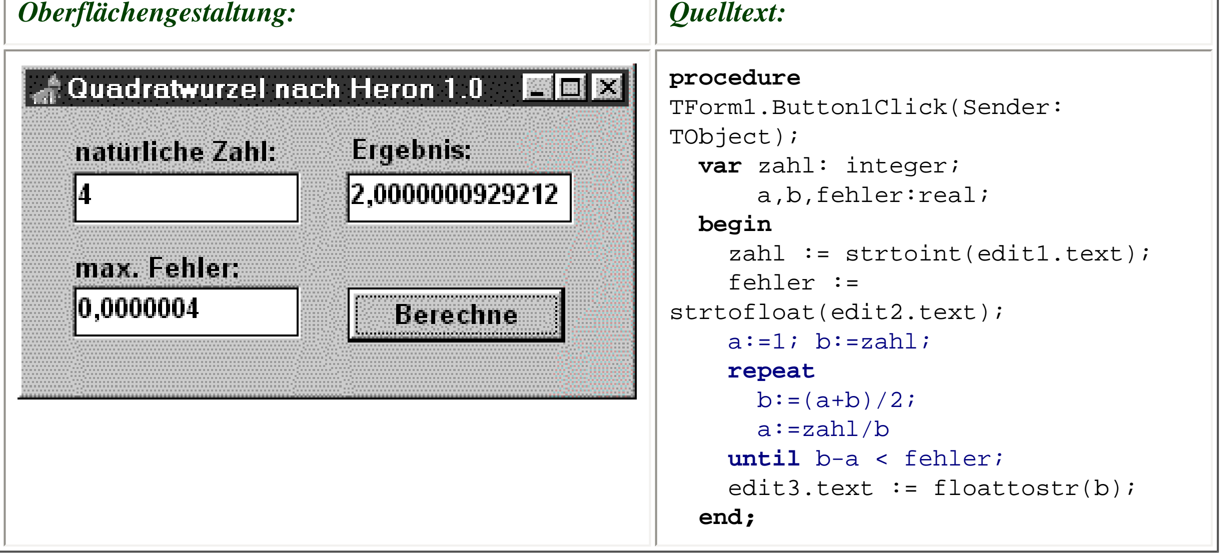
Zur Beachtung:

Die Variable ***fehler*** legt in der Abbruchbedingung die erlaubte Abweichung zwischen ***b*** und ***a*** fest. Wird ***fehler*** z. B. im Rahmen der Eingabe mit 0.0004 belegt, so bewirkt dies eine Genauigkeit des Ergebnisses auf 3 Nachkommastellen.
Der Rechner prüft nach jedem Schleifendurchlauf, ob die sog. **Abbruchbedingung** erfüllt ist (hier: b-a < fehler). Ist diese erfüllt, wird die Schleife verlassen und mit derjenigen Anweisung fortgesetzt, die der Abbruchbedingung folgt. Bei Nichterfüllung der Abbruchbedingung wird der sog. Schleifenkörper erneut durchlaufen.
Wichtig ist es daher, dass über die der Schleife vorangehenden Anweisungen in Verbindung mit dem Schleifenkörper selbst eine Struktur vorliegt, die nach endlich vielen Durchläufen die Abbruchbedingung auch wirklich erreicht.
Andernfalls hätte man eine **"Endlosschleife"** programmiert, die im schlimmsten Fall einen kompletten Rechnerabsturz zur Folge haben könnte.
Spätestens dann stellt man sich die bange Frage: *"Wann habe ich mein Programm eigentlich das letzte Mal gesichert?"* ;-)

Programmierung der Quadratwurzel nach Heron



a) Grundvariante



Oberflächengestaltung:

Quelltext:

```
procedure
TForm1.Button1Click(Sender:
TObject);
  var zahl: integer;
      a,b,fehler:real;
  begin
    zahl := strtoint(edit1.text);
    fehler :=
strtofloat(edit2.text);
    a:=1; b:=zahl;
    repeat
      b:=(a+b)/2;
      a:=zahl/b
    until b-a < fehler;
    edit3.text := floattostr(b);
  end;
```

b) Erweiterte Variante

Bezüglich Benutzerfreundlichkeit und Erkenntnisgewinn lässt die Grundvariante des Programms noch einige Wünsche offen:

- der Heronsche Algorithmus ist nur für natürliche Zahlen **`x>=1`** definiert, das Eingabe-Textfeld erlaubt jedoch auch Einträge von Werten **`x<=0`** (Absturzgefahr / falsches Ergebnis);
  Lösungshinweis: Eingabesicherung mittels Repeat-Schleife

- im mathematischen Sinne ganzzahlige Ergebnisse sind aufgrund ihrer iterativen Erzeugung mit "fehlerhaften" Nachkommastellen belastet;
  Lösungshinweis: Formatierung mittels FloatToStrF

- es ist nicht abzusehen, wie viele Schleifendurchläufe zum Erreichen der geforderten Genauigkeit notwendig waren;
  Lösungshinweis: Zählfunktion in Schleifen

- dem Nutzer müsste erklärt werden, was mit max. Fehler gemeint ist und welche Werte hier als Eingabe sinnvoll sind.
  Lösungshinweis: Eingabefunktion mittels TScrollBar

Aufgabe:

Das Programm ist so zu erweitern, dass möglichst viele der oben aufgeführten Kritikpunkte überwunden werden.

Zur Orientierung kann die nebenstehende Abbildung verwendet bzw. das Programm heron2.exe downgeladen werden.

Bemerkenswert ist die geringe Anzahl an Iterationen, die zum Erreichen hoher Genauigkeiten erforderlich ist. Immerhin ist nach 7 Schritten die Wurzel der Zahl 78 auf 6 Nachkommastellen genau bestimmt!

Anmerkungen:

Um alle Näherungswerte (im Bsp. der Verlauf von b) anzeigen zu können, empfiehlt sich die Verwendung einer **ListBox-Komponenten**. Natürlich muss die "Bestückung" der ListBox *(ListBox1.Items.Add(FloatToStr(b));)* innerhalb der Schleife erfolgen.

Lösungshinweise:

Eingabesicherung mittels Repeat-Schleife

Das Prinzip ist schnell erklärt: der Nutzer ist so oft zur Wiederholung seiner Eingabe aufzufordern, bis sein Eingabewert im erlaubten Bereich liegt - also wieder eine typische Schleifenanwendung!

Beispiel: Es dürfen über Edit1 nur ganzzahlige Werte 10 <= x <= 20 eingelesen werden.

In die entsprechende Ereignisbehandlungsroutine, meist Procedure Button*Click(...); wäre an den Anfang folgende Schleife einzuarbeiten:

```
Repeat
  x := StrToInt (Edit1.Text);
Until (x>=10) AND (x<=20);
```

Nun muss der Nutzer für den Fall einer Fehleingabe diese so oft wiederholen, bis der Eingabewert im "erlaubten" Bereich ist. Nachteil: unter Umständen weiß derjenige gar nicht, *was* er falsch gemacht hat.

Erweiterung: Über ein Dialogfenster soll der Nutzer über seinen Fehler informiert werden; anschließend ist das entsprechende Edit-Feld zu löschen und der Cursor zwecks Neueingabe wieder in diesem zu positionieren.

```
Repeat
  x := StrToInt (Edit1.Text);
  if (x < 10) OR (x>20) then begin
    ShowMessage ('Wert muss zwischen 10 und 20 liegen!');
    Edit1.Clear;
    ActiveControl := Edit1;
  end {of if};
Until (x>=10) AND (x<=20);
```

Um schließlich noch eine Doppeltformulierung des erlaubten / verbotenem Wertebereichs zu vermeiden, empfiehlt sich die Verwendung einer booleschen Variable, im Beispiel `korrekt: Boolean;`

```
Repeat
  x := StrToInt (Edit1.Text);
  if (x >= 10) AND (x<=20) then
    korrekt := true
  else begin
    korrekt := false;
    ShowMessage ('Wert muss zwischen 10 und 20 liegen!');
    Edit1.Clear;
    ActiveControl := Edit1;
  end {of else};
Until korrekt;
```

Formatierung mittels FloatToStrF

Während der Programmierer bei der Verwendung der Funktion "FloatToStr" keinen Einfluss auf das Erscheinungsbild der String-Ausgabe des Real-Wertes hat, eröffnen sich durch "FloatToStrF" vielfältige Formatierungsmöglichkeiten. So lässt sich Beispielsweise die Anzahl der angezeigten Nachkommastellen in Abhängigkeit der iterierten Genauigkeit einstellen. Näheres zur Verwendung von "FloatToStrF" ist der Delphi-Hilfe zu entnehmen.

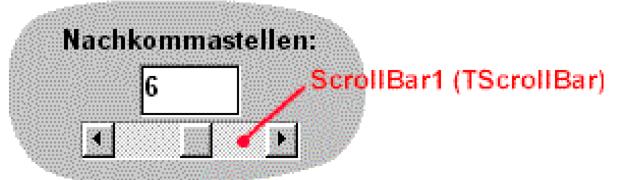
Zählfunktion in Schleifen

So wie Kinder mit den Fingern zählen, kann man's auch den Schleifen "beibringen". Man führe einfach eine ganzzahlige Zählvariable ein, setze deren Wert vor der Schleife auf Null und erhöhe diesen bei jedem Schleifendurchlauf um den Wert 1. Am Ende gibt die Zählvariable Auskunft darüber, wie oft die Schleife durchlaufen wurde.

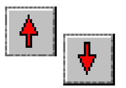
Beispiel:

```
zaehler := 0;
Repeat
  {...}
  zaehler := zaehler + 1;
Until ... ;
```

Eingabefunktion mittels TScrollBar



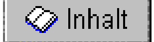
Zur Einstellung der Nachkommastellen und damit des zulässigen Fehlers wurde hier eine Komponente **TScrollBar** verwendet, deren aktueller Wert in dem darüber liegenden Textfeld (TEdit) zur Anzeige gelangt. Um die bei Realtypen verwalteten Nachkommastellen nicht überzustrapazieren, wären über den Objektinspektor für ScrollBar1 die Eigenschaften *Min=1* und *Max=10* zu setzen. Die Eigenschaft *ScrollBar1.Position* gibt den vom Nutzer eingestellten ganzzahligen Wert (zwischen Min und Max) zurück.
Zu beachten ist auch der Zusammenhang zwischen eingestellten Nachkommastellen und dem sich daraus ergebenden zulässigen Fehler. In der Abb. bedeuten z.B. 6 Nachkommastellen einen Fehler von 0.0000004.



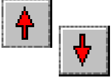
c) Zusatzaufgabe

Heron wird zugeordnet, auch einen Algorithmus zur näherungsweisen Berechnung von Kubikwurzeln entwickelt zu haben. Dies ist sehr nahe liegend, denn was mit Rechtecken und Quadraten funktioniert müsste auch auf Quader und Würfel anwendbar sein ...

1. Transformieren Sie mit einem einfachen Zahlenbeispiel die oben gezeigten Lösungsskizzen auf das Problem der Kubikwurzel und entwickeln Sie die entsprechenden Gleichungen bzw. Wertzuweisungen für die Iteration der Ergebnisses!
2. Verallgemeinern Sie die Lösungsidee zu einem Algorithmus in Struktogrammform.
3. Erstellen Sie ein nutzerfreundliches Delphi-Programm zur iterativen Berechnung der 3. Wurzel einer natürlichen Zahl.



[1. Übertragung Struktogarmm in Quelltext]
[2. "Schreibtischtest" von zyklischen Strukturen]
[2.1. Beispielaufgabe und Lösung]
[2.2. Aufgabe zum Schreibtischtest]
[3. Programmieraufgabe "Eigenwilliger Kredit"]
- [Synchronisation von TListBox-Komponenten]
- [Ausgabe im Währungsformat]



1. Übertragung Struktogramm in Quelltext:

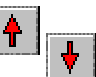
Als algorithmische Kontrollstrukturen kennen wir von den vorangegangenen Seiten ***Sequenzen***, ***Alternativen*** und ***nichtabweisende Zyklen***. Diese lassen sich in Problemlösungsalgorithmen in nahezu beliebiger Weise und Tiefe miteinander kombinieren bzw. ineinander schachteln. Die nachfolgenden Struktogramme sind gemäß Beispiel a) 1:1 in pascalgerechte Notationen zu übersetzen, wobei *"a"* für eine allgemeine Anweisung und *"b"* für eine allgemeine Bedingung steht.

	Struktogramm	Pascal-Quelltext	

a)	Wdh. / a1 / b1 (j: a2, a3 – n: a4) / bis b2	`repeat`
		`  a1;`
		`  if b1 then`
		`    begin`
		`      a2; a3`
		`    end`
		`  else`
		`    a4`
		`until b2;`	

b)	a1 / b1 (j: a2, a3, a4 – n: Wdh. a5, a6 bis b2) / a7	**?**	

c)	Wdh. a1 a2 Wdh. a3 a4 Wdh. a5 a6 bis b3 bis b2 bis b1	**?**	
d)	b1 j n Wdh. j b2 n a1 a2 a3 bis b3 j b4 n Wdh. a4 a5 bis b5 a6 a7 a8	**?**	



2. "Schreibtischtest" von zyklischen Strukturen:

Zweck:

- Erkennen der Variablenbelegungen für verschiedene Eingabewerte,
- Nachweis, ob eine Abbruchbedingung bei bestimmten Eingabewerten erreicht wird oder nicht.

2.1. Beispielaufgabe und Lösung:

a) Zu testen ist folgender nichtabweisender Zyklus für die Eingabe edit1.text := ‘4’ !

```
x := StrToInt(edit1.text);
n := 0; y := 1;
REPEAT
  n := n+1;
  y := y*2
UNTIL n = x;
edit2.text := IntToStr(y);
```

b) Welche mathematische Funktion wird von der Struktur realisiert ?

c) Was würde geschehen, wenn man edit1.text mit ‘-1’ belegt ? - Schlussfolgerung ?

Lösung + Hinweise:

zu a) Der Schreibtischtest erfolgt am besten in Tabellenform. Hier werden alle im Algorithmus relevanten Variablen eingetragen und hinsichtlich der Änderung ihrer Werte beim (gedanklichen) Ablauf des Programmes untersucht. Sind Schleifen im Spiel, so muss nach jedem fiktiven Durchlauf das Erreichen der Abbruchbedingung überprüft werden.

Variablen:		x	n	y
Vorgabe:		*4*	*0*	*1*
Durchläufe:	1.	*4*	*1*	*2*
	2.	*4*	*2*	*4*
	3.	*4*	*3*	*8*
	4.	*4*	*4*	*16*
	Abbruch, da **`n=x`**			

zu b) Hierbei ist die Abhängigkeit der Ausgabevariable(n) von der/den Eingabevariable(n) zu untersuchen, im Beispiel also y = f(x). Erkennt man die Abhängigkeit nicht sofort, führe man weitere "Schreibtischtests" mit geänderten Eingabevariablen durch!

Im Beispiel beträgt $y = 2^x$

zu c) Da n bereits im ersten Schleifendurchlauf mit dem Wert 1 belegt ist und in jedem weiteren Durchlauf um 1 erhöht wird, entstünde bei Eingabe von -1 für x eine sog. "Endlosschleife", weil die Abbruchbedingung **`n=x`** theoretisch niemals erreicht wird.

In der Praxis endet das Ganze entweder mit einer Fehlermeldung (Bereichsüberschreitung / Stacküberlauf etc.) oder mit einem tobsüchtigen Nutzer, der im schlimmsten Fall irgendwann die Reset-Taste betätigt ;-)

Fazit:

1. Insbesondere zyklische Strukturen sollten vor dem Programmstart gedanklich daraufhin getestet werden, welche Eingabewerte zu Endlosschleifen o. a. Fehlern führen könnten.
2. Mit Hilfe einer zu programmierenden Eingabesicherung "zwinge" man den Nutzer dazu, nur "erlaubte" Werte einzugeben.
3. Da 1. und 2. nie 100% Sicherheit bieten, speichere man jedes in Entwicklung befindliche Programm vor dem Compilieren bzw. Starten.

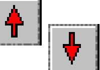
2.2. Aufgabe zum Schreibtischtest

a) Das nachstehende Programmstück ist per "Schreibtischtest" für den Eingabewert ‘5’ zu untersuchen !

```
x := StrToInt(edit1.text);
f := 1;
REPEAT
  f := f * x;
  x := x-1
UNTIL x<1;
edit2.text := IntToStr(f);
```

b) Welche mathematischen Funktion wird hierbei realisiert ?

c) Welcher Wertebereich darf nicht zur Eingabe verwendet werden und weshalb?

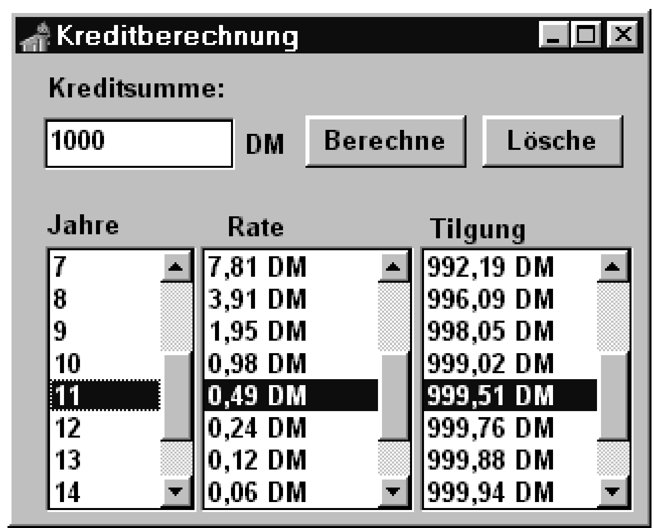


3. Programmieraufgabe "Eigenwilliger Kredit":

a) Ein nicht gewinnsüchtiger Vater mit nennenswerten Ersparnissen trifft mit seinem finanzbedürftigen Sohn eine eigenwillige "Kreditvereinbarung":

Der Sohn erhält einen Kredit in gewünschter Höhe. Es werden keine Zinsen erhoben. Nach einem Jahr soll der Sohn die Hälfte der Kreditsumme zurückzahlen. Nach jedem weiteren Jahr beträgt die Rückzahlung jeweils die Hälfte der Rückzahlung des Vorjahres bis nach n Jahren der Kredit getilgt ist.

Man entwerfe einen Algorithmus (Struktogramm) und schreibe ein Programm, welches für beliebige Kreditsummen bei Rundung auf ganze Pfennige die Anzahl der Jahre ermittelt, die bis zur vollständigen Tilgung benötigt werden sowie die jährlich zu zahlende Rate auflistet.



Benötigte Variablen: *jahre* : ganzzahlig; *kredit, rate, tilgung* : reelle Zahlen.

Nebenstehende Abbildung zeigt einen Vorschlag zur Oberflächengestaltung, wobei die Ausgabe über drei nebeneinander stehende TListBox-Komponenten erfolgt.

Lösungshinweise:

1. Synchronisation von TListBox-Komponenten
2. Ausgabe im Währungsformat

b) Schon eher etwas für Fortgeschrittene:

Eine günstige Stunde ausnutzend, erreicht der Sohn folgende nicht ganz unwesentliche Modifizierung des Kreditvertrages:

Die Abzahlungen können in umgekehrter Reihenfolge stattfinden, d. h. im ersten Jahr erfolgt die Überweisung der kleinsten Rate. Dies steigert sich bis zum letzten Abzahlungsjahr, in dem die Hälfte der ursprünglichen Kreditsumme fällig wird.

Die Problemlösung von a) ist in Struktogramm- und Programmform entsprechend zu modifizieren!

Lösungshinweise:

Synchronisation von TListBox-Komponenten

Falls nebeneinander angeordnete TListBox-Komponenten zusammen gehörende Wertereihen enthalten, erwartet der Nutzer eine gewisse Synchronität der Anzeige. Wird im obigen Beispiel etwa in der ListBox1 auf das Jahr Nr. 11 geklickt, so sollen auch in ListBox2 und ListBox3 die zugehörigen Einträge markiert erscheinen.

Die nachfolgende Prozedur passt den jeweiligen ItemIndex (markierten Eintrag) von ListBox2 und ListBox3 an den in ListBox1 markierten Eintrag an.

```
procedure TForm1.ListBox1Click(Sender: TObject);
  begin
    listbox2.itemindex:=listbox1.itemindex;
    listbox3.itemindex:=listbox1.itemindex;
  end;
```

Ausgabe im Währungsformat

Der Delphi-Hilfe ist unter dem Stichwort "FloatToStrF" u. a. folgendes zu entnehmen:

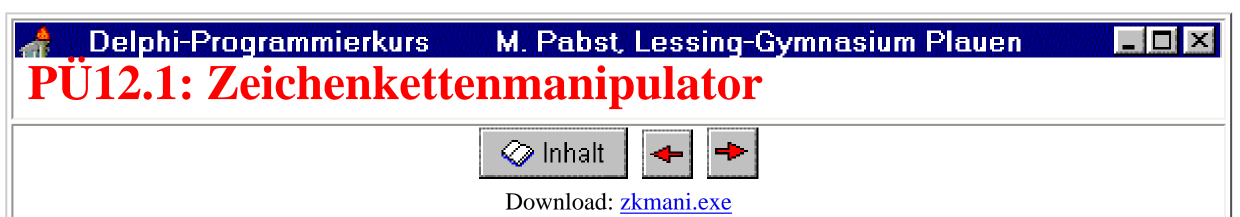
*Funktion **FloatToStrF**(Value: Extended; Format: TFloatFormat; Precision, Digits: Integer): String;*

Beschreibung

FloatToStrF *konvertiert einen durch Value gegebenen Fließpunktwert in seine String-Darstellung.*
Der Parameter Format bestimmt das Format des resultierenden Strings.
Der Parameter Precision bestimmt die Genauigkeit des übergebenen Wertes. Er sollte für Werte vom Typ Single auf 7 oder weniger gesetzt sein, für Werte vom Typ Double auf 15 oder weniger und bei Extended-Werten auf höchstens 18.
Die Bedeutung des Parameters Digits hängt vom gewählten Format ab.

ffCurrency *Währungsformat. Der Wert wird in einen String konvertiert, der eine Währungsbetrag angibt. Die Umwandlung wird durch die globalen Variablen CurrencyString, CurrencyFormat, NegCurrFormat, ThousandSeparator und DecimalSeparator gesteuert, die alle mit den Werten initialisiert werden, die in der Windows-Systemsteuerung, Abschnitt Ländereinstellungen angegeben wurden. Die Anzahl Ziffern nach dem Dezimalpunkt wird durch den Parameter Digits bestimmt und muß zwischen 0 und 18 liegen.*

Im "Kreditprogramm" könnte das Währungsformat wie folgt in die Ausgabeschleife implementiert werden:
```
Listbox2.Items.Add (FloatToStrF (rate, ffCurrency, 8, 2));
```



Bearbeitung von Zeichenketten

Aufgabenstellung:

Im Zuge der Bearbeitung soll ein Formular entstehen, das inhaltlich der nebenstehenden Darstellung entspricht!

Die zu bearbeitende Zeichenkette soll in Edit1 eingegeben werden und nach erfolgter Manipulation dort wieder erscheinen.

Benutzen Sie beim Programmieren die Online-Hilfe von Delphi!

Ereignisbehandlung	Erläuterung, Hilfethemen
Edit1Change	Die momentane Länge des Strings soll in Edit4 angezeigt werden. - *Hilfethema: Funktion* ***"Length"***
ohne erstes	Das erste Zeichen des Strings soll gelöscht werden. - *Hilfethema: Prozedur* ***"Delete"***
ohne letztes	Das letzte Zeichen des Strings soll gelöscht werden.
Lösche Stelle	Das Zeichen an der in Edit2 angegebenen Stelle soll gelöscht werden.
Einfügen	Das in Edit3 eingegebene Zeichen soll an die in Edit2 vermerkte Stelle in den String eingefügt werden. - *Hilfethema: Prozedur* ***"Insert"***
alle groß	Sämtliche Zeichen des Strings, die Buchstaben sind, sollen in Großbuchstaben gewandelt werden. - *Hilfethema: Funktion* ***"Uppercase"***
alles klein	Sämtliche Zeichen des Strings, die Buchstaben sind, sollen in Kleinbuchstaben gewandelt werden. - *Hilfethema: Funktion* ***"Lowercase"***
Anzahl Zeichen	In Edit5 soll ausgegeben werden, wie oft das in Edit3 angegebene Zeichen im String vorhanden ist.
Lösche Zeichen	Alle Zeichen im String, die mit dem in Edit3 angegebenen Zeichen übereinstimmen, sollen gelöscht werden.
Umkehr	Der String soll umgekehrt wieder ausgegeben werden.

Palindrom	Ein Palindrom ist ein Wort, das vorwärts wie rückwärts gelesen die gleiche Buchstabenfolge aufweist (siehe Beispiel!). Wenn dies nach erfolgter Umkehr der Fall ist, soll die Eigenschaft RadioButton1.checked mit true belegt werden, ansonsten wird RadioButton2 true gesetzt.

Beispielprozedur:

```
procedure TForm1.Edit1Change(Sender: TObject);
var kette : String[100];
    laenge: Integer;
begin
  kette := Edit1.Text;
  laenge := Length(kette);
  Edit4.Text := IntToStr(laenge);
end;
```

PÜ 12.2: Transformation zwischen Zahlensystemen

Inhalt

Download: zkonvert.exe

1. Einführung - Szenario
2. Aufgabenstellung
3. Algorithmisch-programmiertechnische Grundlagen
3.1. Wandlung Dual in Dezimal
3.2. Wandlung Dezimal in Dual
3.3. Besonderheiten des Hexadezimalsystems
3.4. Spezielle Pascal-Sprachelemente zum Projekt

1. Einführung - Szenario

Dank seiner Ausstattung mit zehn Fingern ist für den Menschen der Umgang mit dem Dezimalsystem die natürlichste Sache der Welt. Man geht davon aus, dass unsere Vorfahren vor etwa 5000 Jahren begannen, die Dinge, die sie umgaben, zu zählen. Um so schwerer fällt uns naturgemäß der Umgang mit Zahlensystemen, die nicht auf der Basis 10 beruhen, etwa dem Dualsystem oder dem Hexadezimalsystem.

In diesem Projekt geht es daher um die Aufstellung und Programmierung von Algorithmen zur Wandlung des dezimalen Positionssystems in andere Positionssysteme und umgekehrt.

Vorauszusetzendes:

Die Schüler kennen die algorithmischen Grundstrukturen Sequenz, Alternative und Zyklus und sind in der Lage, diese zur Lösung einfacher Problemstellungen miteinander zu kombinieren bzw. zu verschachteln. Kenntnisse zum Datentyp Zeichenkette sowie Fertigkeiten bezüglich Zugriff und Manipulation einzelner Elemente der Kette werden vorausgesetzt.

Projektverlauf:

Zunächst wird die Überführung einer Dezimalzahl in eine Dualzahl und umgekehrt an einfachen Beispielen verdeutlicht. Es werden Algorithmen in Struktogrammform abgeleitet und diese wiederum in eine einfache Oberfläche implementiert.
Anschließend werden die Kenntnisse auf das Problem der Hexadezimalzahlen übertragen und Lösungen zur Behandlung der Hexadezimalziffern > 9 diskutiert.
Schließlich können die Algorithmen des Projektes soweit verallgemeinert werden, dass die Basis selbst zur Variablen wird.

Seitenanfang

2. Aufgabenstellung

Ähnlich der Abbildung ist ein Zahlenkonverter zu programmieren, wobei jedes Textfeld zur Eingabe einer Zahl des angegebenen Zahlensystems genutzt werden kann und auf Betätigung des jeweiligen Go!-Buttons hin die Wandlung in die beiden verbleibenden Systeme erfolgen soll.

Zahlenkonvertierung
Dezimalzahl
go! 45054
Dualzahl
go! 1010111111111110
Hexadezimalzahl
go! AFFE
Lösche
Ende

Für alle Eingaben ist abzusichern, dass keine "unerlaubten" Zeichen zur Verarbeitung gelangen.

Empfohlene Vorgehensweise:

1. Dezimales und duales Positionssystem klarmachen!
2. Algorithmus "Dezimal in Dual" an Zahlenbeispielen sowie im Struktogramm nachvollziehen und danach in die Programmoberfläche (vgl. Abb.) implementieren!
3. Besonderheiten des Hexadezimalsystems herausarbeiten, den Algorithmus "Dezimal in Dual" zu "Dezimal in Hexadezimal" erweitern und implementieren (in die gleiche Prozedur wie 2.)!
4. Programierung der Wandlung "Dual nach Dezimal" - dabei eine Eingabeprüfung auf korrekte Dualzahl einarbeiten! Die Konvertierung "Dual in Hexadezimal" kann, nachdem die entsprechende Dezimalzahl im oberen Edit-Feld ausgegeben wurde, durch Auslösen des Button1.Click-Ereignisses simuliert werden ;-)
5. Die Verfahrensweise von 4. ist auf "Hexadezimal in Dezimal" bzw. "Hexadezimal in Dual" zu übertragen!

Weiterführende Aufgaben:

- Entwicklung eines Zahlenkonverters, der eine einzugebende Zahl beliebiger Basis (2 <= Basis <= 36) in ein Ausgabesystem mit ebenfalls beliebiger Basis wandelt. Der Ziffernvorrat betrage dabei {0...9; A...Z}
- Erweiterung des obigen Konverters für ganze Zahlen auf den Bereich der reellen Zahlen. (Eine Aufgabe für Freaks, denn hier kapituliert sogar der Windows-Taschenrechner! Wer's geschafft hat bitte Mail + Attachement an mich ;-)
- Für die Wandlung von Römischen Zahlen in Dezimalzahlen und umgekehrt ist ein Programm zu entwickeln.

Seitenanfang

3. Algorithmisch-programmiertechnische Grundlagen

3.1. Wandlung Dual in Dezimal

Machen wir uns zunächst nochmals das **Dezimalsystem** (dezimales Positionssystem) klar.
Basis = 10, Ziffern = {0...9}

Beispiel: 1472

10^3 10^2 10^1 10^0

1 4 7 2 $= 2*10^0 + 7*10^1 + 4*10^2 + 1*10^3$

$= 2*1 + 7*10 + 4*100 + 1*1000$

$= 2 + 70 + 400 + 1000$

$=$ **1472**

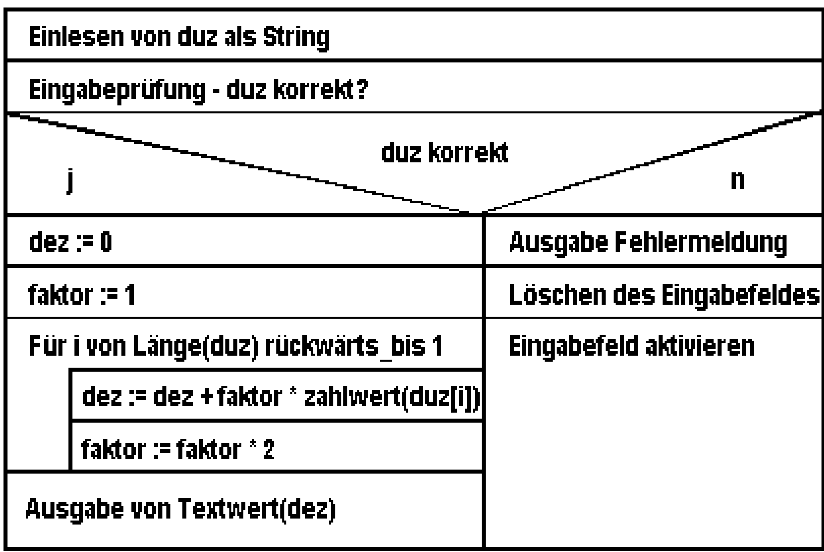
Diese formale Betrachtungsweise wird auf das **Dualsystem** übertragen.
Basis = 2, Ziffern = {0; 1}

Beispiel: $\mathbf{1101_2}$

$$
\begin{array}{lllll}
 & 2^3\ 2^2\ 2^1\ 2^0 & & & \\
\mathbf{1\,1\,0\,1}_2 & = 1*2^0 & + 0*2^1 & + 1*2^2 & + 1*2^3 \\
 & = 1*1 & + 0*2 & + 1*4 & + 1*8 \\
 & = 1 & + 0 & + 4 & + 8 \\
 & = & & \mathbf{13}_{10} &
\end{array}
$$

Ein entsprechender Algorithmus muss also die Zeichenkette, welche die Dualzahl präsentiert, vom letzten bis zum ersten Zeichen durchlaufen und jeweils den Zahlwert der betreffenden Ziffer bilden. (bei Dualzahlen 0 oder 1)
Diese Zahlwerte sind mit den Faktoren der zugehörigen Position zu multiplizieren und zur Dezimalzahl aufzusummieren. (Verdeutlicht wird dieses Vorgehen an der oben grün dargestellten Zeile)

Struktogramm Dual in Dezimal



Realisierung der Eingabeprüfung

Es wird für jedes Zeichen überprüft, ob es sich in der erlaubten Wertemenge befindet. Sollte dies auch nur einmal nicht der Fall sein, ist die Eingabe nicht korrekt. Hilfreich erweist sich in diesem Fall die Verwendung einer boolschen Variablen.

```
...;
{Eingabeprüfung}
korrekt := true;
for i := 1 to Length(duz) do
  if not (duz[i] in ['0'..'1']) then
    korrekt := false;
{Wandlung bei korrekter Eingabe}
if korrekt then
  begin
    ...
  end
{Fehlermeldung bei nicht korrekter Eingabe}
else
  begin
  ...
```

```
end;
```

[Seitenanfang](#)

3.2. Wandlung Dezimal in Dual

Algorithmus:

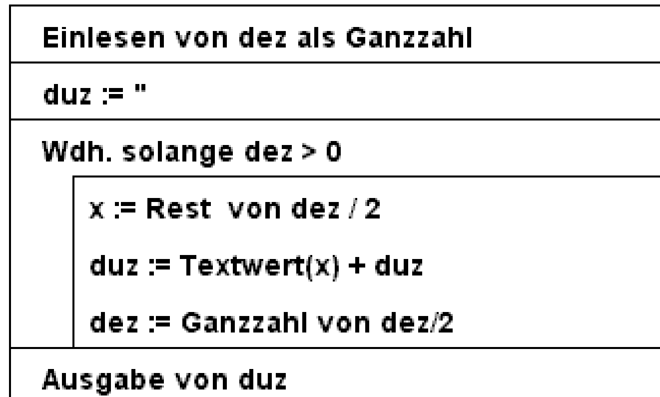
Die Dezimalzahl wird fortlaufend *ganzzahlig* mit 2 dividiert.
Der jeweils entstehende Rest wird von rechts nach links an die Dualzahl angefügt.
Das Verfahren endet, wenn das Ergebnis der ganzzahligen Division 0 beträgt.

23 : 2 = 11 Rest 1
11 : 2 = 5 Rest 1
5 : 2 = 2 Rest 1
2 : 2 = 1 Rest 0
1 : 2 = 0 Rest 1

10111_2

Abb. Wandlung der Dezimalzahl 23 in eine Dualzahl

Struktogramm Dezimal in Dual



[Seitenanfang](#)

3.3. Besonderheiten des Hexadezimalsystems

Da dieses Zahlensystem auf der ***Basis 16*** beruht, sind also auch ***16 verschiedene Ziffernzeichen*** notwendig. Neben den bekannten Ziffernzeichen 0..9 kommen zusätzlich die Buchstaben A..F als "Ziffern" zum Einsatz. (siehe Tabelle)

Dez.	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
Hex.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**	**10**	**11**	**...**

Wandlung Hexadezimal in Dezimal (Zahlenbeispiel):

16^2 16^1 16^0

$$
\begin{aligned}
\mathbf{2\ E\ A}_{16} &= 10*16^0 + 14*16^1 + 2*16^2 \\
&= 10*1 + 14*16 + 2*256 \\
&= 10 + 224 + 512 \\
&= \mathbf{746}_{10}
\end{aligned}
$$

Wandlung Dezimal in Hexadezimal (Zahlenbeispiel):

679	:	16	=	42	Rest	7
42	:	16	=	2	Rest	10
2	:	16	=	0	Rest	2

$2A7_{16}$

Zu beachten ist dabei, dass für Rest-Werte zwischen 10 und 15 die entsprechenden Buchstaben eingesetzt werden müssen.

Im Programm kann dies auf verschiedene Weise realisiert werden:

- mit einer geschachtelten IF-THEN-ELSE-Struktur **:-(**
- mittels CASE-Anweisung **:-)**
- durch Zugriffe auf die ASCII-Tabelle - siehe Chr(x) / Ord(z) **:-)**
- unter Verwendung einer Zeichenkettenkonstante, die alle Hexziffern in gegebener Reihenfolge enthält, wobei der Restwert als Index für die jeweilige Hexziffer fungiert **:-))**

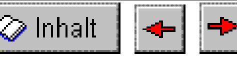
Seitenanfang

3.4. Spezielle Pascal-Sprachelemente zum Projekt

Befehlswort	*Erläuterungen und Beispiele*
DIV	Ganzzahlige Division (Weglassen der Kommastellen) z.B. 7 div 2 => 3
MOD	Restwert der ganzzahligen Division z.B. 7 mod 2 => 1 (denn 7 : 2 = 3, Rest 1)
Longint	Datentyp für lange Integer-Zahlen (32 Bit) Wertebereich (-2147483648 .. 2147483647)
IN	Feststellen einer Mengenzugehörigkeit z.B. if zeichen in ['0'..'9'] stellt fest, ob der Wert von Zeichen innerhalb der Zeichenmenge von 0 bis 9 ist. if buchst in ['a'..'z', 'A'..'Z'] ermittelt, ob buchst mit einem Buchstaben belegt ist
Chr(x)	x ist ein Integer-Ausdruck. Das Ergebnis der Funktion ist ein Zeichen, dessen ASCII-Code dem Wert von x entspricht. Für x sind Werte von 0 bis 255 zugelassen.z.B. chr(68) => 'D'
Ord(z)	z ist ein ASCII-Zeichen. Das Ergebnis der Funktion ist derjenige Integer-Wert, der die Position des Zeichens in der ASCII-Tabelle angebt. z.B. ord('G') => 71

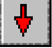
Delphi-Programmierkurs M. Pabst, Lessing-Gymnasium Plauen

PÜ13: Felder, Such- und Sortierverfahren



Downloads: sortiere.com (Zip)
feldsort.exe

1. Einführung - Szenario
2. Aufgabenstellung
3. Algorithmisch-programmiertechnische Grundlagen
    - 3.1. Arrays und Stringlisten
    - 3.2. Arbeit mit dem Zufallsgenerator
    - 3.3. Sequentielles Suchen
    - 3.4. Algorithmen von Sortierverfahren
        - 3.4.2. Sortieren durch Austauschen
        - 3.4.3. Quicksort
    - 3.5. Effizienzvergleich der Sortieralgorithmen



1. Einführung - Szenario

"Wer Sortieralgorithmen programmieren kann, beherrscht das kleine Einmaleins der Programmierkunst!"

Diese These ist nicht unumstritten, weil etliche das Programmieren Lehrende meinen, das kleine Einmaleins ginge noch viel viel weiter. Andererseits hat so mancher von diesen (mich eingeschlossen) seine liebe Not, wenn er beispielsweise "Quicksort" aus dem Stegreif in Programmiersprache niederschreiben soll...

Wie dem auch sei, in der Arbeit mit Schülern bieten sich Sortierverfahren immer wieder an, um Kenntnisse zu algorithmischen Grundstrukturen zu bündeln und, ohne dass dies aufgesetzt wirkt, mit Aspekten der theoretischen Informatik (z.B. aus der Komplexitätstheorie) zu verquicken.

Außerdem zeigt ein Blick in die Geschichte der Rechentechnik, dass neben der Automatisierung des Rechnens an sich vor allem der Wunsch nach Beherrschung und möglichst schneller Bearbeitung größerer Datenmengen eine Haupttriebkraft der Entwicklung automatisierter Datenverarbeitung darstellt.
Denken wir nur an *Hermann Hollerith*, der mit der Entwicklung eines Lochkartensystems die Bevölkerungszählung der USA von 1890 revolutionierte, wodurch deren Auswertungszeit von vorausgesagten zehn Jahren auf damals sensationelle sechs Wochen reduziert werden konnte.

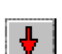
Vorauszusetzendes:

Die Schüler kennen die algorithmischen Grundstrukturen Sequenz, Alternative und Zyklus und sind in der Lage, diese zur Lösung einfacher Problemstellungen miteinander zu kombinieren bzw. zu verschachteln. Kenntnisse zum Datentyp Zeichenkette sowie Fertigkeiten bezüglich Zugriff und Manipulation einzelner Elemente der Kette sollten im Sinne einer Propädeutik zu Feldzugriffen und -operationen hinreichend gefestigt vorliegen.

Projektverlauf:

Anhand einer vorgegebenen Programmoberfläche werden die Schüler mit der komplexen Problemstellung konfrontiert und diese alsdann in überschaubare Einzelaufgaben (Teilprobleme) gegliedert. Die so entstandene Aufgabenfolge bestimmt dann den Unterrichtsverlauf über mehrere Stunden, wobei die Stoffauswahl und -vermittlung sich stets dem zu lösenden Teilproblem unterordnet.

Seitenanfang



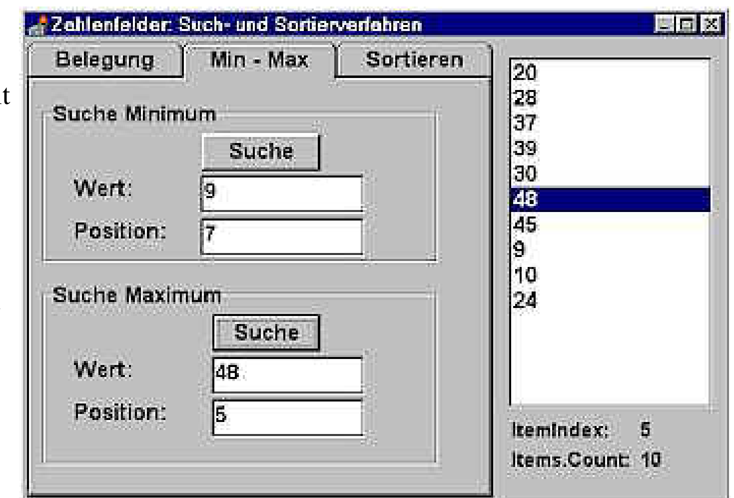
2. Aufgabenstellung

Gegeben ist die abgebildete Programmoberfläche, die grob gesehen aus einem TabbedNotebook mit drei Registerseiten und einer ListBox besteht.

Page1: Belegung der ListBox mit Eingabewerten

1. Über Button1Click (Zahl anfügen) soll ein in Edit1 einzutragender Zahlenwert der Stringliste von ListBox1 hinzugefügt werden. Button3Click (Lösche) soll die Stringliste vollständig leeren.
   Grundlagen: TListBox - das Windows-Listenfeld
2. Das Ereignis Button2Click (Start) soll die Belegung der ListBox mit einer festzulegenden Anzahl von Zufallszahlen realisieren. Die Größe der einzelnen Zufallszahlen soll zwischen Null und einer einzugebenden Obergrenze liegen. Es ist abzusichern, dass nicht mehr als 5000 Zufallszahlen erzeugt werden.
   Grundlagen: Arbeit mit dem Zufallsgenerator

Page2: Sequentielle Suche nach Minimum und Maximum



1. Gesucht ist ein Algorithmus zur Suche des Minimums bzw. Maximums in einer Liste mit n Elementen. Neben den Werten von Min / Max ist auch deren Position in der Liste zu ermitteln und anzuzeigen (Voraussetzung für Sortierverfahren!). Sollte das Minimum bzw. Maximum unter den Listenelementen mehrfach vorhanden sein, ist nur die Position des ersten Auftretens zu speichern.
   Grundlagen: Sequentielles Suchen
2. Die Suchalgorithmen sind in das Programm zu implementieren und mit verschiedenen Belegungen zu testen, ggf. zu korrigieren.
   Beachte: Das Windows-Listenfeld enthält eine Stringliste. Damit also nicht das "alphabetische" Minimum bzw. Maximum ermittelt wird, müssen bei Vergleichsoperationen die Listenelemente in ein Zahlenformat konvertiert werden (z.B. mit der Funktion StrToInt).

Page3: Sortieralgorithmen und deren Effizienz

1. Das Sortieren der ListBox über direkte Zugriffe auf die Stringliste ist zwar möglich, dauert aber durch deren interne Datenstruktur und notwendige Typkonvertierungen schon bei relativ geringen Listengrößen unerträglich lange.
   Daher ist es notwendig, die Daten der Stringliste zunächst in ein Zahlenfeld zu transformieren und nach dem Sortiervorgang die geordneten Feldelemente wieder als Stringliste darzustellen.

   a) Machen Sie Sich (falls nötig) mit den Datentyp "ARRAY" vertraut und stellen Sie Gemeinsamkeiten und Unterschiede zwischen Arrays und ListBox-Komponenten heraus!
   Grundlagen: Arrays und Stringlisten

   b) Deklarieren Sie im Programm als globale Variable ein Feld für 5000 ganze Zahlen.
   Bereiten Sie danach die Prozedur Button6Click (Sortieren durch Austauschen) für das Sortieren vor, indem Sie die Transformationsalgorithmen *Stringliste => Zahlenfeld* und *Zahlenfeld => Stringliste* implementieren!
   Grundlagen: Datentransfers zwischen Stringlisten und typisierten Feldern

2. Anhand des altbewährten Programms sortiere.com (Download als Zip hier möglich) sollen die Algorithmen wesentlicher Sortierverfahren erkannt und verbalisiert werden. Schreiben Sie die Algorithmen so kurz wie möglich und so präzise wie nötig auf, so dass eine x-beliebige Person anhand Ihres Textes eine Menge von 10 bis 15 Zahlenkärtchen algorithmisch fehlerfrei nach dem jeweiligen Verfahren sortieren kann.
   zwei Beispiele: Sortieralgorithmen

3. **a)** Für das Sortieren durch Austauschen sind Struktogramm und Delphi-Quelltext gegeben.
   - Implementieren Sie den Algorithmus in Ihr Programm unter dem entsprechenden Button-Click-Ereignis.
   - Ergänzen Sie den Algorithmus, so dass die Anzahl der Vergleiche und Vertauschungen über Variablen mitgezählt wird und eine Ausgabe erfolgt.

   b) Der Algorithmus für das Verfahren "Bubble-Sort" ist in Struktogrammform zu entwerfen und danach in das Programm zu implementieren. Verfahren Sie weiter wie unter a)!

   c) Der Quicksort-Algorithmus liegt in *verbaler Form* und auch als *"einbaufertiger" Quelltext* vor.
   - Vollziehen Sie den verbalen Algorithmus an der Struktur des Quelltextes nach und beachten Sie dabei die Schachtelung der Prozeduren (farbige Markierung)!
   - Kopieren Sie den Quelltext über die Zwischenablage in Ihr Programm und passen Sie diesen den Gegebenheiten Ihrer Oberfläche an (Komponenten-Bezeichner, Variablen etc.)!

4. Für die nachfolgend durchzuführenden Effizienzvergleiche können Sie Ihr eigenes Programm oder aber die downloadbare Version feldsort.exe nutzen.

   a) Lassen Sie verschieden große Felder aus Zufallszahlen sortieren und füllen Sie folgende Tabelle aus:

	Anzahl der zu sortierenden Elemente					
Sortieraufwand mit:	*500*	*1000*	*2000*	*3000*	*4000*	*5000*
- Sort. durch Austauschen						
- Bubble-Sort						
- Quicksort						

   Hinweis: Nach erfolgtem Sortieren muss die das Feld repräsentierende ListBox zunächst geleert und danach wieder mit neuen Zufallswerten in entsprechender Anzahl gefüllt werden! Ansonsten entstehen unrealistische Ergebnisse.

b) Übertragen Sie die Tabelle in ein Kalkulationsprogramm und stellen Sie den Sortieraufwand in Abhängigkeit der Feldgröße für alle drei Verfahren in einem Liniendiagramm dar!
Beurteilen Sie die Effizienz der untersuchten Sortierverfahren!

c) Angenommen, ein Rechner benötige zum Sortieren von 5000 Elementen durch Austauschen 1 Minute.
- In welcher Zeit wäre das gleiche Feld mittels Quicksort sortiert?
- Wie lange würde der Sortiervorgang für 1 Million Elemente jeweils mit Sortieren durch Austauschen bzw. Quicksort dauern?

Grundlage: Berechnung des Sortieraufwandes

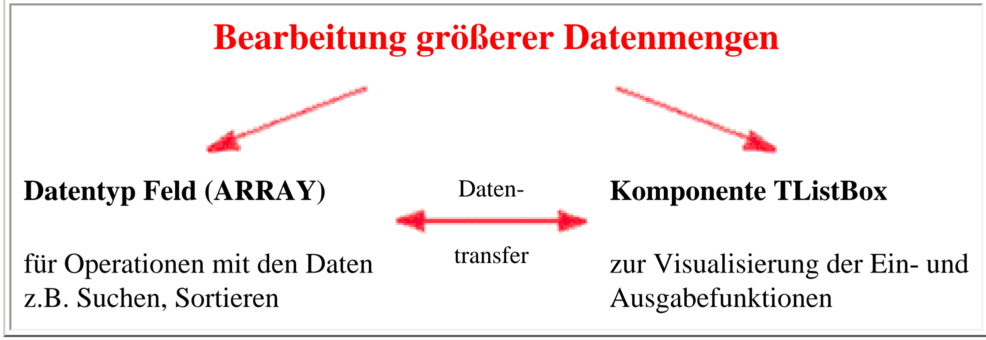
Seitenanfang

3. Algorithmisch-programmiertechnische Grundlagen

3.1. Arrays und Stringlisten

Ziele:

- Kennen lernen des Datentyps Feld (Reihung) zunächst in eindimensionaler Ausprägung,
- Nutzung des Windows-Listenfeldes (TListBox) zur Visualisierung der E/A-Funktionen von Feldern.



Der Datentyp Array (Feld, Reihung)

Begriff:

Ein Datenfeld ist eine Reihung einer bestimmten Anzahl zusammengehörender Daten gleichen Typs, die unter einer einzigen Variablen abgespeichert sind.

Beispiele: a)Feld aus 100 ganzen Zahlen namens *z*

Feldbezeichner[Index]:	`z[1]`	`z[2]`	`z[3]`	`z[4]`	`...`	`z[100]`
Wert:	74	18	2	61	...	23

b)Feld aus 15 Zeichenketten namens ***pers***

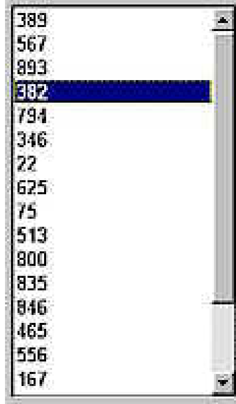
Feldbezeichner[Index]:	`pers[0]`	`pers[1]`	`...`	`pers[14]`
Wert:	'Max'	'Paul'	...	'Felix'

Deklaration: zu a) `var z: ARRAY[1..100] OF Integer;`

zu b) `var pers: ARRAY[0..14] OF String[20];`

Zugriffe: Wie auf andere Variablen kann unter Verwendung des Index auch auf die Elemente eines Feldes mittels Wertzuweisung lesend bzw. schreibend zugegriffen werden.
z.B.
```
x := z[7]; Edit5.Text := IntToStr(z[22]);
pers[1] := 'Paul'; Label8.Caption := pers[14];
```

TListBox - das Windows-Listenfeld



Diese Komponente dient zur Aufnahme und Anzeige einer Stringliste und ist damit auch zur Visualisierung eindimensionaler Datenfelder geeignet.

Wichtige Eigenschaften:

- **Items:** Liste der Strings, die in der ListBox gespeichert sind.
- **Items.Count:** Anzahl der in der Stringliste enthaltenen Elemente.
- **Itemindex:** Ordinalzahl des ausgewählten (markierten) Elementes der Stringliste.

Das erste Listenelement (im Bsp. '389') steht immer an der Position 0! (`ListBox1.Items[0]`). Folglich ergibt sich der Index des letzten Elementes aus `ListBox1.Items.Count-1`.

Beispiele:

1. Anzeige des Index des markierten Listenelementes über Label1:
```
Label1.Caption := IntToStr(ListBox1.ItemIndex);
```

2. Anzeige des letzten Elementes der ListBox über Edit5:
```
Edit5.Text :=
ListBox1.Items[ListBox1.Items.Count-1];
```

- Vereinfachung der Beispiele 1 und 2 mittels With-Operator:
```
with ListBox1 do begin
  Label1.Caption := IntToStr(ItemIndex);
  Edit5.Text := Items[Items.Count-1];
end; {of with}
```

Datentransfer zwischen Stringlisten und typisierten Feldern

Diese Algorithmen beinhalten **Zählschleifen** *(vom ersten bis zum letzten Element tue ...)* und falls es sich

nicht um ein Feld aus Zeichenketten handelt auch **Funktionen zur Typumwandlung**.
Da Stringlisten mit dem Index 0 beginnen, sollten die entsprechenden Felder auch von 0 bis Maxindex deklariert werden, wobei der Maxindex der Anzahl der Listenelemente - 1 entspricht.

Beispiel: Es existiere eine *ListBox1* vom Typ TListBox und ein *zfeld* (Feld ganzer Zahlen) sowie eine ganzzahlige Variable namens *maxindex*.

a) Werte der ListBox in das Feld überführen:

```
maxindex := ListBox1.Items.Count - 1;
for i := 0 to maxindex do
  zfeld[i] := StrToInt(ListBox1.Items[i]);
```

b) Feldelemente in der ListBox anzeigen:

```
ListBox1.Clear;
for i := 0 to maxindex do
  ListBox1.Items.Add(IntToStr(zfeld[i]));
```

Seitenanfang

3.2. Arbeit mit dem Zufallsgenerator

Computer und Zufall

Computer und "echter" Zufall sind so unvereinbar wie Feuer und Wasser. Macht es dem Menschen keinerlei Mühe, z.B. eine Zufallszahl zwischen 1 und 10 anzusagen, so gerät der Rechner hier an seine Grenzen - Computer kennen eben keinen Zufall, ihre inneren Abläufe sind determiniert!

Allerdings gibt es Algorithmen, die bei mehrfacher Ausführung eine Folge von Zahlen erzeugen, die viel mit einer zufällig (z.B. durch Würfeln) erzeugten Zahlenfolge gemeinsam haben. Man nennt jeden Algorithmus dieser Art einen *Zufallsgenerator*. Die Zahlen, die erzeugt werden, heißen *Zufallszahlen* (genauer: *Pseudo-Zufallszahlen*, da tatsächlich nicht etwa gewürfelt, sondern gerechnet wird).
Jeder Zufallsgenerator benötigt eine *Startzahl*, mit deren Hilfe er die erste Zufallszahl berechnen kann, danach wird die zweite berechnet usw. Startete man nun mehrfach mit derselben Zahl, so würde stets dieselbe "Zufallsfolge" erzeugt und der Rechner wäre schnell durchschaut. Beginnt man jedoch mit verschiedenen Startzahlen, so erscheinen völlig verschiedene Zufallsfolgen.

Die folgende Rechenvorschrift stellt einen einfachen Zufallsgenerator dar:

`Xn+1 = (Xn * 473 + 577) MOD 2551;`
`0 <= X0 <= 2550` (Startwert beliebig auswählbar)

mit **`X0 = 123`** entsteht die Zufallsfolge: 83, 1571, 1319, 2020, 512, 408, 2236, ... und
mit **`X0 = 550`** entsteht die Zufallsfolge: 525, 1455, 22, 779, 1700, 1112, 1047, ...

Zufallsgeneratoren existieren in den meisten imperativen Programmiersprachen als vordefinierte Funktion und sind intern sicherlich weit komplizierter aufgebaut, als das angeführte Beispiel.

Der Zufallsgenerator in Pascal:

Der Startwert braucht hierbei nicht vom Nutzer eingegeben zu werden, er wird vielmehr aus der Systemzeit des Rechners gebildet und diese ändert sich wenigstens alle hundertstel Sekunden.

`Randomize;`	Anweisung zur Ermittlung des Startwertes für den Zufallsgenerator
`Random(x);`	Funktion zum Erzeugen einer Zufallszahl im Bereich `0 <= Zahl < x`

Beispiele:

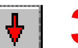
1. Simulation eines Würfels:
```
Randomize;
wurf := Random(6)+1;
```

2. Es sind 1000 Zufallszahlen im Bereich von 0 bis 10000 zu erzeugen und in einer ListBox anzuzeigen.
```
Randomize;
for i := 0 to 999 do
  ListBox1.Items.Add(IntToStr(Random(10001)));
```

3. **zuf** sei als Feld ganzer Zahlen deklariert und soll mit 20 Zufallszahlen im Bereich von 0 bis 100 belegt werden.
```
Randomize;
for i := 0 to 19 do
  zuf[i] := Random(101);
```

   Beachte: Randomize darf nur einmal, und zwar vor der Zählschleife aufgerufen werden, da sonst aufgrund der Geschwindigkeit des Rechners innerhalb 1/100 Sekunde immer wieder derselbe Startwert erzeugt wird, ergo auch gleiche "Zufallszahlen" entstehen.

Seitenanfang



3.3. Sequentielles Suchen

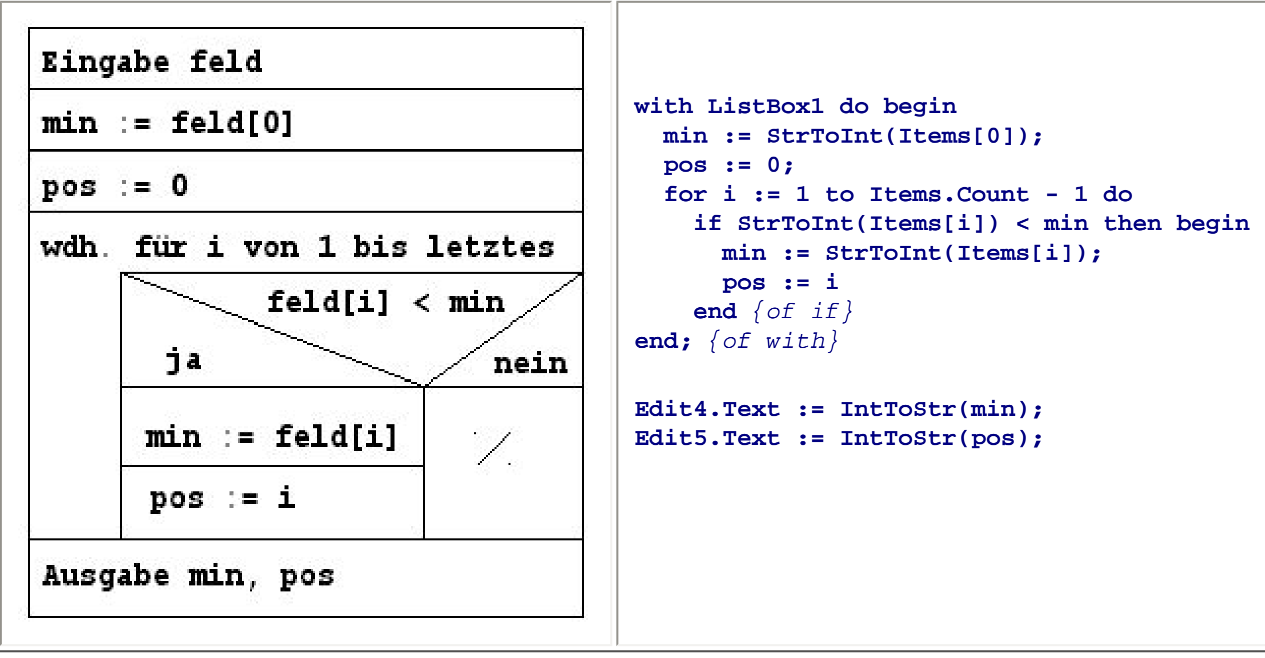
Bei der sequentiellen Suche vergleicht man den Suchschlüssel der Reihe nach mit den Schlüsseln der Feldelemente, beginnend mit dem ersten. Die Suche ist beendet, wenn man ein Element antrifft, dessen Schlüssel gleich dem Suchschlüssel ist, oder wenn man alle Elemente des Feldes verglichen hat, ohne den Suchschlüssel anzutreffen.

Bei der Suche nach dem Minimum oder Maximum eines Feldes stellt sich die Frage nach dem Suchschlüssel anders, denn man weiß ja vorher nicht, welchen Wert das Minimum bzw. Maximum hat.

Für die Suche nach dem Minimum modifiziert sich obiger Algorithmus daher wie folgt:
Man merke sich den Wert des ersten Feldelementes und vergleiche diesen mit den zweiten. Falls das zweite Element kleiner ist, merke man sich dessen Wert und vergleiche anschließend mit dem dritten usw. bis man alle Feldelemente verglichen hat.

Neben dem Wert des Minimums an sich ist meist auch noch dessen Position im Feld bzw. in der Liste von Interesse (besonders im Hinblick auf das spätere Sortieren).

Struktogramm Minimum-Suche *(in einem Feld ganzer Zahlen)*	**Object-Pascal-Text der Minimum-Suche** *(in einer ListBox, Stringliste aus Zifferzeichen)*



Seitenanfang

3.4. Algorithmen von Sortierverfahren

Sortieren ist nicht gleich Sortieren!

Meist sortiert man eine Datenmenge mit dem Ziel, später in dieser nach bestimmten Kriterien effektiv suchen und auswählen zu können. Denken wir z.B. an ein Telefonbuch: Eine vereinfachte Modellierung der Datenobjekte könnte wie folgt aussehen:

Offensichtlich sind für zielgerichtetes Sortieren die Sortierkriterien und deren Hierarchie entscheidend.

```
Sortiere Telefonbuch nach:
  - Anschrift.Ort
    - Name.Nachname
       - Name.Vorname
          - Anschrift.Straße
             - Anschrift.Hausnummer
                - Rufnummer
```

Das Sortieregebnis könnte dann so aussehen:

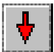
Plauen

...	...	...
Müller, Falk	Birnenweg 6	314279
Müller, Frank	Apfelbaumweg 11	227495
Müller, Frank	Zwetschgenweg 34	368812
Müller, Frank	Zwetschgenweg 34	368942
Müller, Frederike	Kastanienweg 41	734519
...	...	...

Wenngleich auch der Sortiervorgang einfacher wäre, so würde es dem gewöhnlichen "Leser" eines Telefonbuches gar wenig nützen, hätte man dieses primär nach Rufnummern sortiert ;-)

Neben der Auswahl und Hierarchisierung der Suchkriterien bei komplexen Datenobjekten spielen auch die Sortieralgorithmen eine wesentliche Rolle. Es gibt einfach zu programmierende, die bei einer größeren Datenmenge unerträglich lange dauern und komplizierter zu programmierende, die flink sind wie die Wiesel!

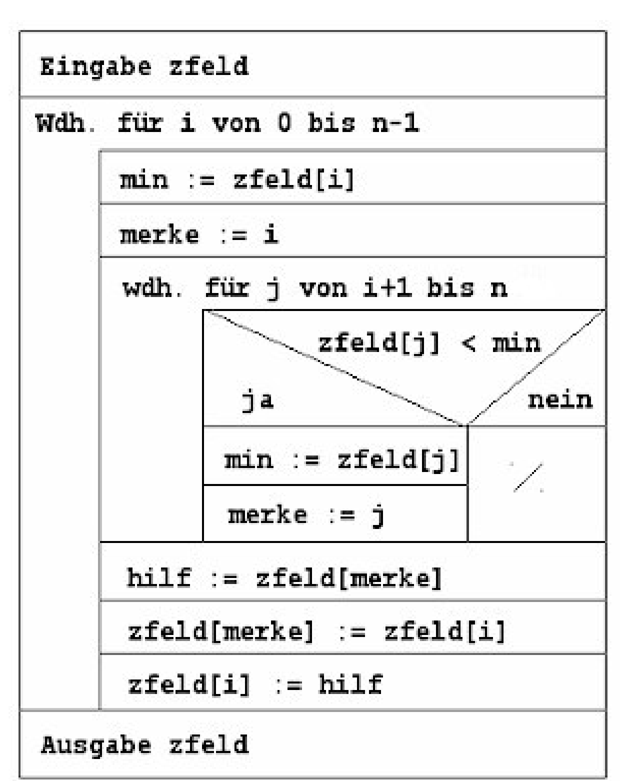
Die nachfolgenden beiden Sortieralgorithmen beziehen sich auf ein eindimensionales Feld ganzer Zahlen namens *zfeld* mit *n* Elementen.

 3.4.2. Sortieren durch Austauschen

Beschreibung:

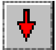
Beim ersten beginnend wird der Reihe nach jedes Feldelement mit all seinen Nachfolgern verglichen und sich der jeweils kleinste Nachfolger gemerkt. Ist dieser kleiner als das betreffende Feldelement, so werden beide ausgetauscht.

Struktogramm:



Delphi-Prozedur:

```
procedure TForm1.Button6Click(Sender: TObject);
{Sortieren durch Austauschen}
var min, n, i, j, merke: integer;
begin
{Eingabe Zahlenfeld (n = Anz. der Feldelemente)}
n := ListBox1.Items.Count - 1;
...
{Sortiervorgang}
for i := 0 to n - 1 do begin
  min := zfeld[i];
  merke := i;
  for j := i+1 to n do
    if zfeld[j] < min then begin
      min := zfeld[j];
      merke := j;
    end {of if};
  hilf := zfeld[merke];
  zfeld[merke] := zfeld[i];
  zfeld[i] := hilf;
end; {of for}
{Ausgabe Zahlenfeld}
...
end;
```

 Quicksort

Beschreibung:

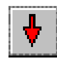
Das Feld wird über ein Trennelement aus der Feldmitte in zwei Teilfelder zerlegt. Alle Elemente, die größer sind als das Trennelement, werden rechts von diesem und die kleineren links davon abgelegt. Die entstandenen Teillisten werden ebenso behandelt bis keine weitere Zerlegung mehr möglich ist. (Rekursives Vorgehen!)

Delphi-Prozedur:

Bezüglich obiger Aufgabenstellung ist die nachfolgende Prozedur "einbaufertig", sofern gleiche Komponenten und gleiche Bezeichner verwendet wurden. Zwecks späterem Effizienzvergleichs werden die Anzahl der Vergleiche und Vertauschungen mitgezählt und über Labels ausgegeben. Zur besseren Erkennbarkeit der Schachtelung der eingebauten Unterprogramme sind zusammengehörende Prozedurköpfe und -rümpfe jeweils in der gleichen Farbe dargestellt.
Es scheint sich die obige Bemerkung zu bestätigen, dass schnelle Sortieralgorithmen nicht ganz unkompliziert zu programmieren sind ;-)

```
procedure TForm1.Button8Click(Sender: TObject);
  procedure quicksort(var a: feldtyp; erstes, letztes: integer);
    {Sortieren mittels Quicksort}
    var linker_merker, rechter_merker : integer;
    procedure zerlege (var a: feldtyp; var links, rechts: integer);
      var pivot: integer;
      procedure tausche (var x, y: integer);
        var hilf: integer;
        begin
          hilf := x; x := y; y := hilf;
          vertausche := vertausche + 1;
        end; {of vertausche}
      begin {zerlege}
        pivot := a[(links + rechts) DIV 2];
        repeat
          while a[links] < pivot do begin
            links := links + 1;
            vergleiche := vergleiche + 1;
          end;
          while a[rechts] > pivot do begin
            rechts := rechts - 1;
            vergleiche := vergleiche + 1;
          end;
          if links <= rechts then begin
            tausche (a[links], a[rechts]);
            links := links + 1; rechts := rechts - 1;
            vergleiche := vergleiche + 1;
          end;
        until links > rechts;
      end;{of zerlege}
    begin {quicksort}
      linker_merker := erstes; rechter_merker := letztes;
      zerlege (a, linker_merker, rechter_merker);
      if erstes < rechter_merker then
        quicksort (a, erstes, rechter_merker); {Rekursiver Selbstaufruf!}
      if linker_merker < letztes then
        quicksort (a, linker_merker, letztes); {Rekursiver Selbstaufruf!}
    end; {of quicksort}
  begin {Button8Click}
    vergleiche := 0;
    vertausche := 0;
    screen.cursor := crhourglass;
    lies_zahlfeld_ein;
    quicksort(z, 0, listbox1.items.count - 1);
    gib_zahlfeld_aus;
    screen.cursor := crdefault;
    label15.caption := floattostr(vergleiche);
    label16.caption := floattostr(vertausche);
    edit10.text := floattostr(vergleiche + vertausche);
 end;
```

Seitenanfang



3.5. Effizienzvergleich der Sortieralgorithmen

Sortieren durch Austauschen

Wenn man den vom Rechner zu betreibenden Aufwand beim Sortieren auf die Anzahl der durchzuführenden Vergleichsoperationen beschränkt, so verrät der Algorithmus selbst den Aufwand, der für seine Abarbeitung notwendig ist.

"Beim ersten beginnend wird der Reihe nach jedes Feldelement mit all seinen Nachfolgern verglichen ..."

Da das erste Element n-1 Nachfolger hat, das zweite n-2 Nachfolger usw. ergibt sich folgende Tabelle:

Durchläufe	Vergleiche
1.	n-1
2.	n-2
3.	n-3
...	...
n-2.	3
n-1.	2
n. (letzter)	1

Es lässt sich unschwer erkennen, dass die durchschnittliche Anzahl der Vergleiche pro Durchlauf bei $n/2$ liegt. Da aber n Durchläufe erforderlich sind, ist der Aufwand $A_A(n) \sim n*n/2$, also $A_A(n) \sim ½ n^2$ und letztlich: $A_A(n) \sim n^2$

Folglich liegt eine quadratische Abhängigkeit vor, d.h. es wächst der Sortieraufwand mit dem Quadrat der Feldgröße!

Da nicht jeder Rechner gleich schnell arbeitet, muss für die Ermittlung des zeitlichen Aufwandes eine rechnerspezifische Zeitkonstante, nennen wir sie c, eingeführt werden:

$$A_A = ½ n^2 * c$$

Die Zeitkonstante c ist für den eigenen Computer bestimmbar, indem man für eine geeignete Feldgröße die Sortierzeit mit der Stoppuhr (oder programmiertechnisch) ermittelt und dann obige Formel benutzt.

Beachte: *Die Zeiten zum Einlesen bzw. zur Ausgabe der Feldelemente über die ListBox dürfen natürlich nicht der Sortierzeit zugeschlagen werden. Beim zum Download angebotenen Programm wird daher die eigentliche Sortierzeit über ein rotes Einfärben des zur Aktivitätsanzeige genutzten Textfeldes verdeutlicht.*

Quicksort

Die Herleitung des Sortieraufwandes über den Algorithmus dieses Verfahrens würde den Rahmen der vorliegenden Seite sprengen. Daher sei an dieser Stelle nur die Formel angegeben, die bei guter Näherung die Abhängigkeit $A_Q(n)$ beschreibt:

$$A_Q(n) \sim n * lg(n)$$ bzw. mit Zeitkonstante:

$$A_Q(n) = c * n * lg(n)$$

Der "Traum" einer linearen Abhängigkeit der Sortierzeit von der Feldgröße geht also auch hier nicht in Erfüllung (genauso wenig wie der Traum vom Perpetuum mobile!)
Dennoch zeigt sich die haushohe Überlegenheit dieses innovativen Sortierverfahrens gegenüber den einfacheren mit quadratischer Abhängigkeit!

Beispiel:

Elemente	**Sortieren durch Austauschen**	**Quicksort**

100	$A_A(100)=100^2/2*c$ $A_A(100)=$**5000c**	$A_Q(100)=100*lg(100)*c = 100*2*c$ $A_Q(100)=$**200c**
10000	$A_A(10000)=10000^2/2*c$ $A_A(10000)=$**50000000c** $A_A(10000)=5*10^7c$	$A_Q(10000)=10000*lg(10000)*c = 10000*4*c$ $A_Q(10000)=$**40000c** $A_Q(10000)=4*10^4c$