

# Applikations- entwicklung mit Delphi



## Kursunterlagen

Kursunterlagenversion: 1.1

Softwareversion: 2.0 E

**Autoren: Dr. Silvia Rothen  
Max Kleiner**

1 Einleitung.....	8
1.1 Kursziel .....	8
1.2 Strategischer Hintergrund .....	8
1.2.1 Neues aus Borland .....	8
1.2.2 OLE jetzt noch flexibler .....	9
1.2.3 Client/Server inside .....	9
1.2.4 Kompaktere neue Umgebung .....	10
1.2.5 Projektverwaltung .....	11
1.2.6 Delphi 2 in drei Versionen.....	11
1.2.7 Unterschied zwischen Delphi und Delphi Client/Server .....	12
1.2.8 Portierung von bestehenden Anwendungen.....	13
1.2.9 Delphi 2 mit Object Inspector, Editor und einem Datenbank Form.....	14
1.2.10 Highlights von Delphi 2 .....	14
1.3 Voraussetzungen .....	15
1.4 Einführungsfragen.....	15
1.5 Konventionen im Skript .....	16
1.5.1 Formatierungskonventionen .....	16
1.5.2 Piktogramme.....	16
1.5.3 Delphi-Namenskonventionen.....	17
2 Installation und Handhabung.....	19
2.1 Installation und Laden der Entwicklungsumgebung .....	19
2.1.1 Versionen und Varianten von Delphi .....	19
2.1.2 Die Installation .....	20
2.1.3 Die Entwicklungsumgebung laden .....	21
2.2 Die Delphi-Oberfläche .....	23
2.2.1 Wichtige Einstellungen der IDE .....	25
2.3 Sichtbare Elemente der Entwicklungsumgebung .....	25
2.3.1 Das Formular .....	25
2.3.2 Der Objektinspektor .....	26
2.3.3 Die Komponentenpalette .....	29
2.3.4 Der Quelltexteditor .....	30
2.3.5 Die Symbolleiste .....	32
2.4 Die Projektverwaltung .....	33
2.5 Die Dateien eines Projekts .....	34
2.6 Der integrierte Debugger.....	37
2.7 Rekapitulation .....	38
2.7.1 Kapiteltest.....	38
2.7.2 Übung .....	39
3 Programme und Units.....	40
3.1 Ein Beispielprogramm .....	40

3.2 Die Syntax eines Programms.....	42
3.2.1 Die Syntax des Beispielprogramms.....	42
3.2.2 Die allgemeine Syntax eines Programms.....	44
3.2.3 Reihenfolge der Deklarationen.....	45
3.3 Die Struktur einer Unit.....	45
3.3.1 Die Unit uClose des Beispielprogramms.....	45
3.3.2 Die Syntax einer Unit.....	46
3.3.3 Die Teile einer Unit.....	47
3.4 Indirekte und zirkuläre Referenzen.....	49
3.4.1 Indirekte Unit-Referenzen.....	49
3.4.2 Zirkuläre Unit-Referenzen.....	49
3.5 Rekapitulation.....	50
3.5.1 Zusammenfassung.....	50
3.5.2 Kapiteltest.....	50
4 Visuelle Programmierung.....	51
4.1 Die Philosophie von Windows-Applikationen.....	51
4.2 Formulardesign.....	52
4.2.1 Eine Komponente auf einem Formular plazieren.....	52
4.2.2 Mehrere Komponenten plazieren.....	53
4.2.3 Komponenten löschen.....	55
4.2.4 Gruppieren von Komponenten.....	55
4.2.5 Owner und Parent.....	57
4.3 Ereignisbehandlungsroutinen.....	57
4.4 Die wichtigsten Komponenten.....	59
4.4.1 Die Komponenten des Registers Standard.....	60
4.4.2 Die Komponenten des Registers Additional.....	61
4.4.3 Die Komponenten des Registers Win95.....	62
4.4.4 Die Komponenten des Registers Data Access.....	63
4.4.5 Die Komponenten des Registers Data Controls.....	63
4.4.6 Die Komponenten des Registers Win 3.1.....	64
4.4.7 Die Komponenten des Registers Dialogs.....	64
4.4.8 Die Komponenten des Registers System.....	65
4.4.9 Weitere Register und Komponenten.....	66
4.5 Das Editor-Projekt.....	66
4.6 Modale und nichtmodale Dialogfenster.....	68
4.7 Rekapitulation.....	69
4.7.1 Kapiteltest.....	69
4.7.2 Übung.....	70
5 Konstanten und Variablen.....	71
5.1 Daten und Routinen.....	71
5.2 Grundelemente von Object Pascal.....	72

5.2.1 Tokens .....	72
5.2.2 Operatoren .....	72
5.2.3 Ausdrücke .....	72
5.3 Konstanten .....	73
5.3.1 Begriffsdefinition .....	73
5.3.2 Konstantendeklaration .....	73
5.4 Variablen .....	75
5.4.1 Begriffsdefinition .....	75
5.4.2 Variablendeklaration .....	76
5.4.3 Variablenzuweisungen .....	76
5.5 Typisierte Konstanten .....	78
5.6 Rekapitulation .....	80
5.6.1 Kapiteltest .....	80
6 Typen .....	81
6.1 Definition und Klassifikation von Typen .....	81
6.2 Einfache und String-Typen .....	83
6.2.1 Integer-Typen .....	83
6.2.2 Boolesche Typen .....	84
6.2.3 Der Char-Typ .....	85
6.2.4 Aufzählungs- und Teilbereichstypen .....	85
6.2.5 Realtypen .....	86
6.2.6 String-Typen .....	87
6.2.7 Pascal-Strings und nullterminierte Strings .....	90
6.3 Strukturierte Typen .....	91
6.3.1 Benutzerdefinierte Typen .....	91
6.3.2 Arraytypen .....	92
6.3.3 Mengentypen .....	93
6.3.4 Recordtypen .....	94
6.4 Zeigertypen .....	97
6.5 Kompatibilität von Datentypen .....	98
6.5.1 Typen- und Zuweisungskompatibilität .....	98
6.5.2 Fundamental- und generische Typen .....	99
6.6 Rekapitulation .....	100
6.6.1 Kapiteltest .....	100
7 Ablaufsteuerung .....	102
7.1 Anweisungen, zusammengesetzte Anweisungen und Blöcke .....	102
7.1.1 Anweisungen .....	102
7.1.2 Zusammengesetzte Anweisungen .....	102
7.1.3 Blöcke .....	103
7.1.4 Gültigkeitsbereiche .....	103
7.2 Die If-Anweisung .....	104

7.3 Die Case-Anweisung.....	106
7.4 Schleifen-Programmierung .....	108
7.4.1 Die For-Schleife .....	108
7.4.2 Die While-Schleife .....	110
7.4.3 Die Repeat-Schleife.....	111
7.5 Rekapitulation .....	111
7.5.1 Übung7_1 .....	111
8 Prozeduren und Funktionen .....	112
8.1 Einführung.....	112
8.2 Parameter .....	113
8.3 Lokale Deklarationsteile .....	115
8.4 Prozeduren und Funktionen.....	117
8.4.1 Unterschiede zwischen Prozeduren und Funktionen.....	117
8.4.2 Implementierung von Prozeduren.....	117
8.4.3 Implementierung von Funktionen .....	118
8.4.4 Prozedur- und Funktionsaufrufe .....	119
8.4.5 Deklaration im Interface-Teil.....	119
8.4.6 Eine eigene Funktion erstellen .....	120
8.5 Forward- und External-Deklarationen .....	122
8.5.1 Forward-Deklaration .....	123
8.5.2 External-Deklaration .....	123
8.6 Rekapitulation .....	123
8.6.1 Übung8_1 .....	123
9 Klassen und Instanzen .....	124
9.1 Objektorientierte Programmierung .....	124
9.2 Begriffsdefinitionen .....	126
9.2.1 Klassen, Felder und Methoden.....	126
9.2.2 Instanzen .....	126
9.2.3 Objekte .....	126
9.2.4 Eigenschaften .....	127
9.2.5 Komponenten und OOP .....	127
9.3 Die Syntax einer Klasse .....	128
9.4 Methoden .....	130
9.4.1 Methoden allgemein .....	130
9.4.2 Konstruktoren und Destruktoren.....	132
9.4.3 Eine eigene Suchmethode erstellen .....	133
9.5 Die Klasse TApplication .....	135
9.6 Vererbung .....	136
9.7 Virtuelle und dynamische Methoden .....	136
9.7.1 Polymorphie.....	136
9.7.2 Virtuelle Methoden und late binding .....	137

9.8 Rekapitulation .....	139
9.8.1 Kapiteltest .....	139
10 Eigenschaften und Windows-Botschaften .....	140
10.1 Was sind Eigenschaften (Properties)? .....	140
10.1.1 Vorteile von Properties .....	140
10.1.2 Properties selbst deklarieren .....	141
10.1.3 Standardwerte von Properties .....	143
10.1.4 Erzeugen von Array-Properties .....	144
10.1.5 Ereignisse sind auch Properties .....	144
10.2 Windows Nachrichten .....	145
10.2.1 Windows API Funktionen .....	145
10.2.2 API-Funktionen nach Gebieten aufgeteilt: .....	146
10.2.3 Windows-Messages .....	147
10.2.4 Was enthält eine Windows-Botschaft? .....	150
10.2.5 Eigene Botschaften versenden .....	150
10.2.6 Überschreiben von Botschaften .....	151
10.3 Definieren eigener Botschaften .....	151
10.3.1 Deklarieren eines eigenen Botschaftsbezeichners .....	152
10.3.2 Generelle Botschaftsbearbeitung .....	153
10.4 Rekapitulation .....	154
10.4.1 Zusammenfassung .....	154
10.4.2 Kapiteltest .....	154
10.4.3 Übung .....	154
11 Exception Handling und RTTI .....	156
11.1 Was sind Ausnahmebehandlungen? .....	156
11.1.1 try finally Construct .....	157
11.1.2 try except Construct .....	158
11.1.3 Nochmaliges Auslösen einer Exception .....	159
11.1.4 Verschachtelte Exceptions .....	160
11.1.5 Die vier Ausnahmezustände im Überblick .....	160
11.2 Die Exception-Klasse .....	161
11.3 Run Time Type Information .....	162
11.3.1 Der Operator is .....	162
11.3.2 Der Operator as .....	163
11.4 Klassenreferenzen .....	164
11.4.1 Klassenreferenztypen .....	164
11.5 Rekapitulation .....	165
11.5.1 Zusammenfassung .....	165
11.5.2 Kapiteltest .....	165
11.5.3 Übung .....	166
12 Programmstruktur .....	167

12.1 Entwurf der Struktur .....	167
12.2 Gültigkeitsbereich von Bezeichnern .....	168
12.2.1 Sichtbarkeit von Routinen .....	168
12.2.2 Sichtbarkeit von Blöcken .....	169
12.2.3 Sichtbarkeit von Units .....	169
12.2.4 Sichtbarkeit von Interface- und Standardbezeichnern .....	170
12.2.5 Sichtbarkeit von Klassen .....	170
12.3 Variablen-Typen .....	172
12.3.1 Statische Variablen .....	172
12.3.2 Dynamische Variablen .....	173
12.4 Offene Parameter .....	173
12.4.1 Offene String-Parameter .....	173
12.4.2 Offene Array-Parameter .....	174
12.5 Rekapitulation .....	176
12.5.1 Zusammenfassung .....	176
12.5.2 Kapiteltest .....	176
12.5.3 Übung .....	177
12.6 Schlussprojekt .....	177
13 Begriffsverzeichnis Deutsch - Englisch .....	179
14 Glossar .....	180
15 Index .....	186

## **1 Einleitung**

### **1.1 Kursziel**

Delphi bietet sowohl eine benutzerfreundliche visuelle Entwicklungsplattform als auch eine mächtige objektorientierte Programmiersprache. Deshalb lassen sich mit Delphi nicht nur in kürzester Zeit kleine Windows-Applikationen entwickeln, die gegenüber interpretierten Sprachen beträchtliche Geschwindigkeitsvorteile aufweisen, sondern der objektorientierte Ansatz erlaubt es auch, diese kleinen Programme zu immer grösseren und komplexeren Applikationen weiterzuentwickeln.

Am Ende des Kurses sind Sie mit der Handhabung der Delphi-Entwicklungsumgebung vertraut und kennen die wichtigsten Elemente von Object Pascal, so dass Sie fähig sind, kleinere Windows-Applikationen zu entwickeln.

### **1.2 Strategischer Hintergrund**

Der in diesem Unterkapitel 1.2 folgende Artikel bietet einen strategischen Überblick, was Delphi 2 als Entwicklungsumgebung bietet.

Die meisten Anbieter von Entwicklungsumgebungen rüsten nun auf die 32-Bit-Welt um. Neben dem gravierendsten Fortschritt, dem Wegfall der 64K-Datensegmentgrenze, nützt Borland die Umstellung, um Delphi 2 mit neuen Features und Komponenten abzurunden.

#### **1.2.1 Neues aus Borland**

Borland hat jetzt eine umfassende Strategie für ihre Client/Server-Aktivitäten bekanntgegeben. Dazu gehören neue Produkte und strategische Partnerschaften, mit denen das Unternehmen seine Stellung auf dem Client/Server-Markt untermauern will. Anlässlich der Presseveranstaltung «Borland Spring Tour» in Zürich vom März 96 stellte Borland auch das Kernprodukt ihrer Strategie vor, eben die Hochleistungs-Entwicklungsumgebung Delphi 2. Delphi 2 soll Entwicklern alle Client/Server-Fähigkeiten bieten, die sie für die sichere und schnelle Applikationsentwicklung benötigen. Es erlaubt die problemlose Skalierbarkeit von Desktop- zu unternehmensweit einsetzbaren Lösungen, eben „The Upsiz-



ing Company“. Nun zu den Features. Neben visueller Programmierung wurde in Delphi ein neues Klassenmodell integriert, welches unter dem Namen „Object Pascal“ eine neu strukturierte Version von Pascal darstellt. So unterstützt Delphi 2 das Win95 Interface und das Win95 GUI. Die verbesserte 32-Bit-Borland Database Engine beschleunigt den Datenbankzugriff. Auch eine 32-Bit-Version des Local Interbase Server soll demnächst erscheinen.

#### **1.2.2 OLE jetzt noch flexibler**

Der von Microsoft über die PC-Grenzen hinaus propagierte OLE-Standard zum Austausch von Daten und Funktionen, berücksichtigt man unter Delphi 2 besonders. Die meisten Änderungen führen zu einer noch einfacheren und logischeren Handhabung der OLE-Komponente. Auch die Funktionen der zugehörigen Unit liessen sich überarbeiten und in Methoden von `T OleContainer` umwandeln. Jedoch die Komponentenunterstützung der VCL ist auf OLE-Client-Anwendungen beschränkt. Der Bau eines OLE-Servers (ab Delphi 2 möglich) ist zu komplex, als dass ein paar Komponenten genügen. Um eine Server-Anwendung zu schreiben, müssten Sie die Schnittstellen des Common Object Modells verwenden und eigene COM-Klassen schreiben. Der Entwickler verfügt trotzdem über eine erweiterte Funktionalität, die es erlaubt, fremde Programme aufzurufen und sie aus der Delphi-Applikation heraus zu benutzen. Ähnlich wie Microsoft setzt auch Borland auf die neue Generation der wiederverwendbaren Komponenten `OleCustomControls` (OCX). So ist es z.B. kein Problem eine Zeitreihe aus einer Paradox 7 Datenbank mit Delphi 2 als Frontend in Excel 7 aufzubereiten.

#### **1.2.3 Client/Server inside**

Ein weiteres wichtiges Feature von Delphi ist wie gesagt die Unterstützung von Client/Server-Anwendungen. Aus diesem Grunde ist auch das Entwicklungssystem in drei Versionen erhältlich: in einer Standard Version, einer selbständigen Developer Version mit der Local Interbase Engine und einer Client/Server-Version gebündelt mit dem eigenen Datenbankserver Interbase 4.0 für Windows NT und Netware, zusätzlich die SQL-Links zur Ankopplung an die Datenbankserver von Oracle (das Oracle von Delphi), Sybase/Microsoft und Informix. Von der Konzeption her ist es nicht verwunderlich, dass die Borland-Entwickler eine ei-

gene Datenbank-Engine integriert haben. Sie enthält die BDE-Technologie, die Zugriff auf die bereits erwähnten Datenbankserver ermöglicht. Durch die Einbindung von Microsofts ODBC-Schnittstelle hat man Zugriff auf weitere Datenbanken wie z.B. FoxPro und Access. All das macht Delphi 2 meines Erachtens nach zu einem fast konkurrenzlosen Produkt.

Ein weiteres Highlight soll auch der Import von Klassen aus DLLs sein, von denen dann weitere Unterklassen abgeleitet werden können. Die Unterstützung der alten und neuen Standards wie DDE und OLE 2.0 gehört ebenfalls zum Umfang des Systems. Konkurrenzlos ist auch die Tatsache, dass der in Delphi integrierte Compiler „reine“ EXE-Dateien erzeugt, die dann keinen Interpreter in Form einer oder mehrerer DLLs mehr brauchen. Beeindruckend ist auch die Ausführungsgeschwindigkeit der Programme.

#### **1.2.4 Kompaktere neue Umgebung**

Zu den Hauptbestandteilen der Entwicklungsumgebung in Delphi 2 gehört in erster Linie die sogenannte Speedbar, die dem Entwickler in Form von Speedbuttons nicht nur die wichtigsten Funktionen zur Projektverwaltung, sondern auch Steuerelemente verschiedener Kategorien zur Verfügung stellt. Die Menüstruktur hat Borland grundlegend überarbeitet. Neben neuen Einträgen liessen sich auch viele Unterpunkte neu zusammenfassen und strukturieren. Unter dem Menüpunkt <File/New> lassen sich die Befehle nun kompakter abrufen. Besonders das «abgeleitete Formular» ist neu. So können Formulare als Elternobjekt dienen - mit der Besonderheit, dass, wenn sich die Komponenten auf dem Elternobjekt ändern oder verschieben, auch die Komponenten auf dem abgeleiteten Formular sich automatisch mitverschieben. Bei der Alltagsarbeit häufig benötigte Funktionen zur Komponentenentwicklung und Datenbankpflege ordnete man eigenen Menüpunkten zu, wobei thematisch verwandte Funktionen jetzt zusammengefasst sind.

So kann man zwischen den Standardelementen (Menü, Schalter, Liste u.a.), den Zusatz-Steuerelementen (Timer, Tabelle, Spreadsheet u.a.), den grafischen Elementen, den Dialogfenster-Elementen, den Datei- und Verzeichnis-Auswahlboxen, den Datenbank-Steuerelementen und den (gleich bin ich fertig) durch Visual Basic beliebt gewordenen VBX-Controls wählen. Die Gruppen sind durch Hinzufügen neuer Elemente beliebig erweiterbar, wodurch dem erfahrenen Entwickler das Zukaufen von zusätzlichen Controls

(obwohl möglich) von Drittanbietern erspart bleibt. Doch Delphi bietet noch wesentlich mehr. Jedes der Standardelemente ist sowohl in 2D als auch in 3D verfügbar. Wird ein Element auf einer Form platziert, so wird der zugehörige Quellcode automatisch erzeugt. Jedes eingefügte Steuerelement ist eine echte Instanz (Objekt) der zugehörigen Klasse und der Entwickler hat somit mit Hilfe des Objektinspektors die Möglichkeit, von dieser alten Klasse eine neue Klasse abzuleiten, die Eigenschaften und Methoden des Vorfahren in der Klassenhierarchie zu vererben und diese durch zusätzliche zu ergänzen. Diese objektorientierten Features machen Delphi 2 zu einem sehr offenen und flexiblen System.

#### **1.2.5 Projektverwaltung**

Auch die Art der Projektverwaltung wie auch die visuelle Programmierung ist bei Delphi sehr durchdacht. Zu jedem Projekt gehört wie üblich eine Projektdatei. Es können eine oder mehrere Units wie auch Formen eingebunden werden. Die Units beinhalten übrigens wie bisher die klassische Dreiteilung in Interface, Implementation und Initialisierung. Auf Wunsch kann auch mit MDI-Fenstern oder auch ganz ohne Formulare gearbeitet werden. Einer meiner Tests bestand darin, so viele MDI wie möglich zu generieren um zugleich den lokalen Heap zu beobachten. Die resultierende Kompaktheit des Speicherlayouts war beeindruckend. Die Einstellungen der Optionen für die Projekte ist in Delphi sehr detailliert. Wie in Borland Pascal lassen sich zahlreiche Optionen ein und ausschalten, unter anderem solche Compilerschalter wie wort- bzw. byteweise Ausrichtung der Daten, Prüfung von Laufzeitfehlern und Verwendung von Callback-Funktionen. Zu den Linker-Optionen gehören in erster Linie die Einstellung der Heap- und Stackgröße, welche die dynamische Speicherverwaltung einer EXE-Datei erst ermöglichen.

#### **1.2.6 Delphi 2 in drei Versionen**

Delphi 2 ist wie gesagt in drei Versionen erhältlich, nämlich Standard (ca. 299.- Upgrade), Developer (ca. 399.- Upgrade) und die Client/Server Suite (ca. 1499.- Upgrade). Ich möchte hier näher auf den Leistungsumfang der C/S-Suite eingehen. Mit der Delphi 2 C/S-Suite erhalten Sie ein komplettes 32-Bit Client/Server Entwicklungssystem, das Ihnen einen vollständig skalierbaren Datenbankzugriff ermöglicht.

Entwicklern bietet dies folgende Vorteile: Entwickeln Sie im Team mit dem integrierten Version Control System (Intersolv PVCS). Mit dem SQL Database Explorer haben Sie die Möglichkeit, Server Parameter direkt aus der Oberfläche zu ändern. Sie erhalten einen Überblick über Ihre Datenbanken, Stored Procedures und Triggers. Der SQL Monitor ist in der Lage, Interaktionen zwischen ihrer Anwendung und einem SQL Server zu überwachen. Dadurch können Sie SQL Anwendungen testen und ein Feintuning durchführen.

Beim Upsizing Ihrer Datenbank hilft Ihnen der DataPump Experte. Dieser Experte ist in der Lage, Datenbanken zwischen Formaten und Plattformen zu skalieren. Der Visual Query Builder ist ein visuelles Tool, mit dem Sie SQL Abfragen übersichtlich erstellen können. Dabei erstellen Sie mit dem VQBuilder automatisch fehlerfreie (manchmal ist ein wenig Feinarbeit schon noch vonnöten) ANSI SQL 92 Befehle. Für den Zugriff auf verschiedene Server stehen Ihnen High-Performance 32-Bit SQL Links zur Verfügung. 16-Bit Entwickler erhalten mit der Delphi 2 C/S Suite sowohl die 16-Bit Version als auch neue 16-Bit SQL Links.

Ferner erhalten Sie mit der Suite einen kompletten Handbuchsatz zum Thema Delphi und Entwicklung. Delphi 2 C/S ist mit diesem Lieferumfang ein abgeschlossenes Entwicklungssystem für Client/Server Profis und Entwickler.

#### **1.2.7 Unterschied zwischen Delphi und Delphi Client/Server**

Delphi 2 Client/Server ist für Client/Server und professionelle Datenbankentwickler gedacht. Die Client/Server Version enthält alle Komponenten des Delphi Developer und Standard Paket. Zusätzlich enthält Delphi Client/Server die folgenden Komponenten:

SQL Links Ver. 2.5 sind dabei. Diese Version enthält native Treiber (Hochleistungstreiber) für Oracle, Sybase, MS SQL Server, Informix und Interbase. Diese Links sind weder im Delphi-Paket enthalten noch sind sie einzeln erhältlich. Sie ermöglichen „...unlimited royalty-free deployment across servers“, d.h. man darf beliebig viele Clients, welche auf beliebig viele Server zugreifen können, erstellen ohne dabei Lizenzgebühren an Borland entrichten zu müssen.

Dies gilt übrigens nur für die Anwendung, die von Delphi erzeugt wurde und nicht für Delphi selbst! Wenn ein Team von 10 Entwicklern zusammen auf ein Client/Server Projekt

hinarbeiten, müssen 10 Kopien der SQL Links sprich 10 Kopien von Delphi C/S gekauft werden.

Durch das Local Interbase Deployment Kit kann der Delphi C/S Entwickler die Local Interbase Engine kostenlos mit seiner Anwendung vertreiben ohne irgendwelche Lizenzgebühren zu zahlen. Die Local Interbase Engine unterstützt nur 1 Client.

Teamentwicklung mit check-in, check-out, (Verwaltung von Entwicklungsprojekten im Netz) und Versionskontrolle (PVCS Ver. 5.1 von InterSolv erforderlich)

Die SQL-Version von ReportSmith (ReportSmith SQL) ist dabei.

Der Visual Query Builder (Visual-Tool für Generierung von Queries und SQL Statements) und die Quelltexte der Visual Component Library.

Zielgruppe für Delphi Standard und Developer sind demnach Windows-Entwickler und Programmierer lokaler Datenbank Anwendungen.

#### **1.2.8 Portierung von bestehenden Anwendungen**

In den meisten Fällen wird das Portieren von 16-Bit Delphi Anwendungen auf 32-Bit reibungslos ablaufen. Dennoch gibt es einige Punkte zu beachten:

Änderungen im Windows API lassen sich bei den meisten Programmen von der VCL abfangen. D.h., auf normale Delphi Applikationen wird dies keinen Einfluss haben. Sollten Sie jedoch ein spezielles API ansprechen, dass in Win95 /NT nicht vorhanden ist, müssen sie diesen Bereich manuell anpassen.

Für den gleichzeitigen Umgang mit Integer-Werten bei 16- und 32 Bit (unterschiedlicher Speicherbedarf der Integer-Werte von je 2 bzw. 4 Bytes) enthält der Bereich Integer-Datentypen weitere Informationen. Zudem sollten Sie beachten: Die Grösse von Integer ohne weitere Angaben ist betriebssystemabhängig und kann sich auch in zukünftigen Versionen von Delphi ändern. Am besten ist es, SIZEOF(Integer) zu benutzen, um die aktuelle Grösse des Datentyps zu ermitteln.

Wenn Sie planen, beide Plattformen (Win 16 Bit und Win 32 Bit) parallel zu unterstützen, sollten Sie Ihre Anwendung zuerst unter Delphi 16-Bit Version entwickeln und dann gegebenenfalls um neue 32-Bit Features erweitern oder einfach

nur mit Delphi 2 neu kompilieren. Aber Achtung: Sie können Delphi 1.x Anwendungen zwar in Delphi 2 laden und bearbeiten, umgekehrt kann es jedoch zu Ressourcenkonflikten zwischen beiden Versionen kommen.

#### **1.2.9 Delphi 2 mit Object Inspector, Editor und einem Datenbank Form**

Delphi 2, preisgekrönt durch die Zeitschrift BYTE als „Best Technology of Comdex“, ist meiner Meinung nach das einzige Entwicklungstool für Windows, das die hohe Leistungsfähigkeit eines optimierenden Native-Code-Compilers und integrierten Client/Server-Tools in sich vereint. Das objektorientierte, komponentenbasierende Design von Delphi bietet unbegrenzte Erweiterungsmöglichkeiten und erlaubt es, selbstentwickelte Komponenten ebenso wiederzuverwenden wie Komponenten der Visual Component Library, externe DLLs und bald OBX-Controls. Mit Delphi stellen Entwickler in Rekordzeit lizenzfreie Windows-Anwendungen fertig, die erheblich schneller und stabiler als interpretierter Code sind.

#### **1.2.10 Highlights von Delphi 2**

- 32-Bit Native-Code Compiler, dadurch z.T. deutlich schnellere und kleinere Programme
- hochoptimierender Codegenerator, wie er auch in der kommenden Version von Borland C++ enthalten ist
- kompiliert Programme für Windows 95 und Windows NT
- Volle Unterstützung des Windows 95 User interface
- Erstellung von OBJ-Files
- neue VCL-Komponenten für Win95 Controls
- Unterstützung von OLE Controls (OCX) sowie OLE Automation
- Unterstützung der erweiterten Features von Windows 95 inkl. Multi-Threading
- Schnelle 32-Bit Borland Database Engine und SQL Links
- Überarbeitete und verbesserte Dokumentation

### **1.3 Voraussetzungen**

Eine produktive Kursteilnahme kann nur gewährleistet werden, wenn Sie einerseits mit der Bedienung von Windows als Anwender/in vertraut sind und andererseits auch Grundkenntnisse in der Programmierung haben. Zwar werden alle Grundkonzepte der Sprache Object Pascal, beispielsweise Schleifen, noch einmal eingeführt, aber in relativ raschem Tempo.

Mit Hilfe der nachfolgenden Fragen können Sie deshalb Ihre Vorkenntnisse wiederholen. Der Kursleitung hilft dieser Einleitungstest, Stärken und Schwächen der Teilnehmer/innen abzuschätzen, um besser auf die Bedürfnisse jedes einzelnen eingehen zu können. Für eine fruchtbare Kursteilnahme ist es nicht notwendig, dass Sie alle Fragen beantworten können.

### **1.4 Einführungsfragen**

1. Was ist der Unterschied zwischen kompilierten und interpretierten Programmiersprachen?
2. Welche Arten von Schleifen kennen Sie aus Ihrer bisherigen Programmiererfahrung?
3. Was ist der Unterschied zwischen einer Funktion und einer Prozedur?
4. Warum gibt es lokale und globale Variablen?
5. Kennen Sie den Unterschied zwischen einem Call by Value und einem Call by Reference?
6. Wozu dient die Unterscheidung zwischen einem formalen und einem aktuellen Parameter?
7. Erklären Sie den Begriff „Konstruktor“?
8. Welche Klassenhierarchien kennen Sie?
9. Wozu dienen die rund 700 API-Funktionen in Windows?
10. Was ist ein Fenster in Windows?
11. Welchen Unterschied gibt es zwischen dem Wasserfallmodell und dem Spiralmodell im Software-Engineering?

## 1.5 Konventionen im Skript

Bevor wir in das eigentliche Thema einsteigen können, müssen Sie sich mit einigen Konventionen dieses Skripts vertraut machen. Diese Konventionen betreffen Formatierung, Symbole und Namenskonventionen für Delphi-Code.

### 1.5.1 Formatierungskonventionen

Die typografischen Konventionen entsprechen jenen, die Sie aus neueren Kursunterlagen von Digicomp bereits kennen. Für neue Kursbesucher/innen sind sie hier zusammengefasst:

Menüs wie FILE und andere Elemente der Entwicklungsumgebung, z.B. das Register PROPERTIES, ausserdem auch die Bezeichnungen der einzelnen Properties im Objektinspektor wie NAME, sind alle an den Kapitälchen erkennbar.

Die Formatierung `Code` wird für Listings von Programmcode verwendet oder für Bezeichnungen wie `form1`, die von der Entwicklungsumgebung vergeben werden und Teil des Programmcodes sind.

Ihre eigenen *Eingaben*, die Sie bei Übungen über die Tastatur oder durch die Auswahl von Listefeldern machen müssen, sind dagegen *mit dieser kursiven Formatierung* gekennzeichnet. Tastaturkürzel und Funktionstasten wie CTRL + C oder F1 erkennen Sie an den Grossbuchstaben. **Fett** formatiert ist Wichtiges, beispielsweise die Einführung eines neuen Begriffs. Ordner- und Dateinamen wie «uClose.pas» stehen in speziellen Anführungszeichen.

### 1.5.2 Piktogramme

Zusätzlich zu den Formatierungen kennzeichnen Piktogramme in der Randspalte bestimmte Textteile. Folgende Symbole kommen vor:



Lernziel



Einführung eines neuen Begriffs



Wichtige Information



Codebeispiel



Analyse eines Codebeispiels





## Übung

Diese optischen Hilfsmittel erleichtern Ihnen später, bestimmte Lerneinheiten rasch wieder zu finden.

### 1.5.3 Delphi-Namenskonventionen

Um Programmcode transparenter zu machen, hat es sich eingebürgert, Bezeichner gemäss gewissen Konventionen zu vergeben. Typen werden beispielsweise immer mit einem grossen T begonnen. Buchstabenkürzel geben Hinweise auf den Typ einer Variablen oder einer Komponente. Weil Object Pascal nicht kontextsensitiv ist, kann man auch in die Gross- und Kleinschreibung gewisse Informationen verpacken. Reservierte Wörter werden beispielsweise immer klein geschrieben. Bei Bezeichnern dagegen schreibt man das Typenkürzel am Anfang klein, während man den eigenen Teil des Namens mit einem Grossbuchstaben beginnt. Da die Bezeichner relativ lang sein dürfen, empfiehlt es sich, trotz der zusätzlichen Tipparbeit möglichst sprechende Bezeichner auszuwählen.

Hier werden nur die gebräuchlichsten, in den Übungen verwendeten Kürzel erwähnt.

<b>Kürzel</b>	<b>Komponente/Typ</b>	<b>Beispiel</b>
arr	Array	arrMatrix
btn	Schaltfläche (Button)	btnFarbwechsel
btnbit	Bitmap-Schalter	btnbitClose
btnradgrp	Komponente Radiogroup	btnradgrpWoher
byt	Byte-Zahl	bytPosition
car	Kardinalzahl	carPosition
chr	Char	chrZeichen
chx	Checkbox-Komponente	chxGanzesWort
dlg	Dialogkomponente	dlgSuchen
frm	Formular	frmCloseApp
int	Integer-Zahl	intAnzahlPersonen
lbl	Label	lblHelloWorld
mem	Memo-Komponente	memEditor
mnu	Menükomponente	mnuHauptmenu
mnuit	Menü-Item	mnuitLaden
pnl	Panel-Komponente	pnlButtons
ptr	Pointer	ptrAdresse
rec	Record	recPerson
real	Realzahl	realDistanz
set	Menge	setZeichen
str	String	strSuchtext

Auch Dateinamen werden mit dem Anfangsbuchstaben gekennzeichnet. Projektdateien (.dpr) beginnen mit p, z.B. «pApp2\_1.dpr», und Units (.pas) mit u, z.B. «uClose.pas».

Ein wichtiges Mittel, um Programmcode transparenter zu machen, sind Einrückungen. Code, der anderem Code untergeordnet ist, z.B. Anweisungen innerhalb einer Schleife, wird um eine Position eingerückt. Auch hier existieren verschiedene Konventionen. Anhand der Beispiele und der Erläuterungen im Kapitel über Blöcke werden Sie das im Skript verwendete System rasch begreifen.

## **2 Installation und Handhabung**



Dieses Kapitel macht Sie mit Installation und Handhabung der Delphi-Entwicklungsumgebung vertraut. Am Ende dieses Kapitels wissen Sie,

- welche Versionen von Delphi es gibt,
- wie man Delphi installiert,
- aus welchen sichtbaren und unsichtbaren Elementen die Delphi-Oberfläche besteht,
- wie man die wichtigsten Elemente der Entwicklungsumgebung handhabt
- und welche Dateien zu einem Projekt gehören.

Da Learning by Doing am besten im Gedächtnis haften bleibt, werden Sie bereits Ihr erstes Windows-Programm mit Delphi erstellen.

### **2.1 Installation und Laden der Entwicklungsumgebung**

#### **2.1.1 Versionen und Varianten von Delphi**

Delphi gibt zur Zeit in der Version 1.02 für die Entwicklung von 16-Bit-Programmen (Varianten Standard und Client/Server) und in der Version 2.0 für die Entwicklung von Programmen für Windows-95 oder NT. Die neue 32-Bit-Version liefert Borland in drei verschiedenen Varianten aus:

- Delphi 2 Standard ist eine vollständige visuelle Entwicklungsumgebung, die auf der objektorientierten Sprache Object Pascal basiert. Mit den enthaltenen 32-Bit-Komponenten lassen sich einfache Windows-Anwendungen im Drag-and-Drop-Verfahren erstellen. Für Windows-3.1-Entwicklungen wird die 16-Bit-Version Delphi 1.0 mitgeliefert.
- Delphi 2 Developer enthält über die Standardversion hinaus einige Werkzeuge, die sie für grössere professionelle Projekte, zum Beispiel Datenbanken, besonders geeignet machen. Für Datenbankberichte steht Report Smith zur Verfügung, und ein Installations-Experte erleichtert die Weitergabe Ihrer Applikationen. Für Komponentenent-

wickler unentbehrlich sind Handbuch und Quellcode der visuellen Komponentenbibliothek.

- Die Delphi 2 Client/Server Suite erweitert die vorangehende Version zu einem echten 32-Bit-Client/Server Entwicklungssystem für den beliebig skalierbaren Datenbankzugriff.

Während die Sprache Object Pascal selbst immer in Englisch definiert ist, gibt es die Entwicklungsumgebung mit Hilfesystem und Handbüchern in verschiedenen Sprachversionen. Da auf der CD mit der deutschen Version auch die englische enthalten ist, empfiehlt es sich, die Version in der Landessprache zu kaufen, auch wenn Sie die englische Version installieren möchten. Sie können dann nämlich im Verzeichnis «Runtime», das eine vollständige installierte Version enthält, zusätzlich auch auf die **Online-Hilfe** in Ihrer Muttersprache zugreifen.

#### 2.1.2 Die Installation

Die vorangehenden Ausführungen zu Versionen und Varianten von Delphi waren notwendig, weil sich die Installation je nach Variante etwas unterscheidet. Sie ist aber weitgehend selbsterklärend. Folgende Schritte müssen Sie für die Installation durchführen:

1. Starten Sie Windows 95 oder NT, falls Sie sich nicht bereits darin befinden.
2. Delphi 2.0 wird nicht mehr auf Disketten ausgeliefert, sondern nur noch auf CD-ROM. Legen Sie also die Delphi-CD in Ihr CD-ROM-Laufwerk ein.
3. Wenn Sie jetzt mit dem **Windows-Explorer**, der in Windows 95 den Dateimanager ersetzt, die Verzeichnisse der CD betrachten, ergibt sich für die Developer-Version beispielsweise das Bild in Abb. 1.
4. Öffnen Sie für die deutsche Version den Ordner «Install» oder für die englische «Instalus» und doppelklicken Sie die Datei «Setup».
5. Folgen Sie den Anweisungen des Installationsprogramms.

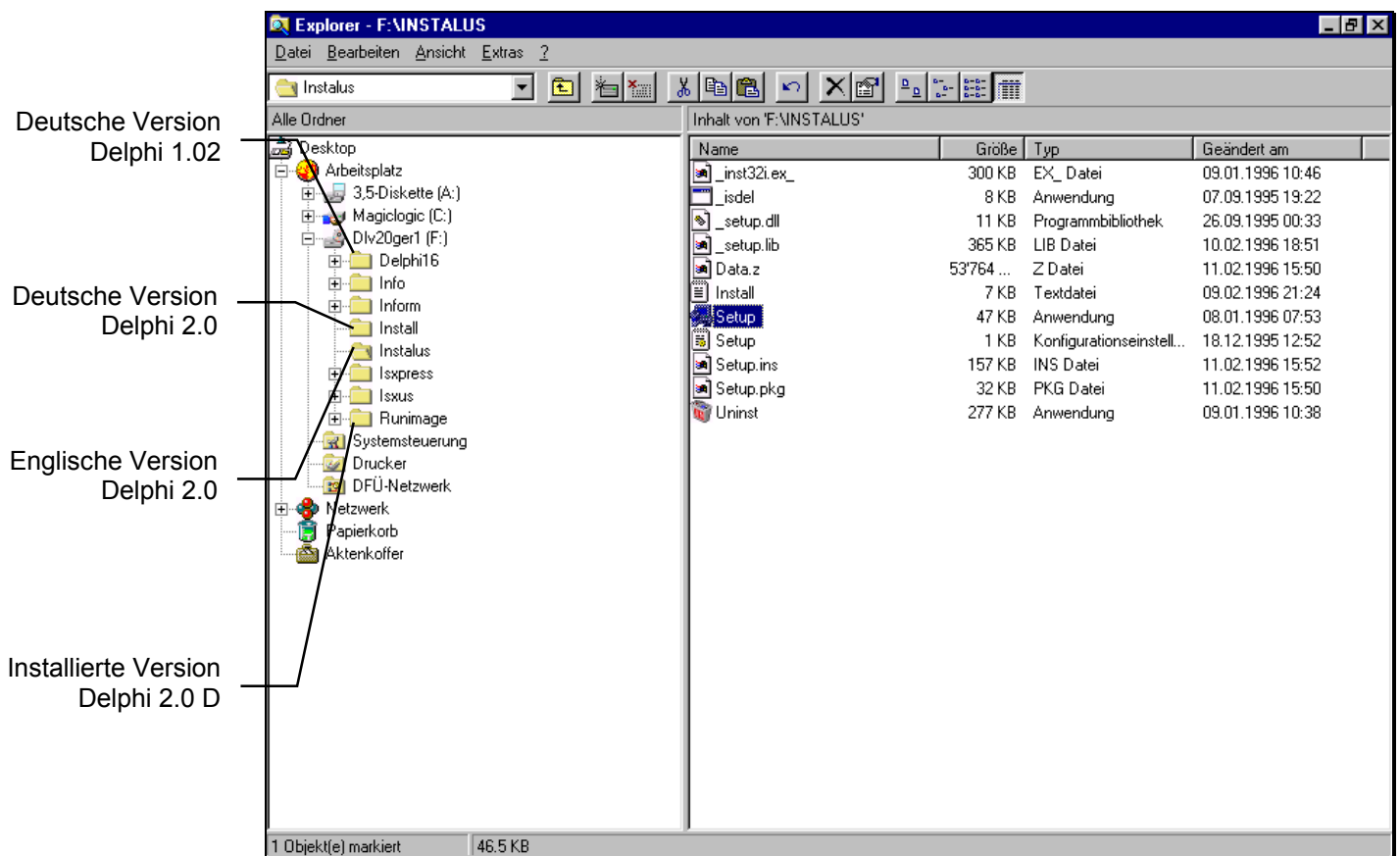


Abb. 1 Inhalt der deutschen CD-ROM Delphi 2.0 Developer

Als erstes zeigt Ihnen das Installationsprogramm die Datei «install.txt» an, die wichtige Informationen zur Unterstützung Ihrer Installation enthält. Dann erstellt es die notwendigen Verzeichnisse und kopiert die Dateien von der Installations-CD oder den Installationsdisketten auf die Festplatte. Am Ende der Installation wird die Datei «readme.txt» angezeigt und auf Ihrer Festplatte gespeichert. Unter anderem finden Sie dort Hinweise, welche Dienstleistungen des Kundendienstes Sie bei Problemen beanspruchen können, und welche weitere Dokumente in der installierten Version zur Verfügung stehen.

### 2.1.3 Die Entwicklungsumgebung laden

Sie können Delphi über den Explorer beziehungsweise unter NT über den Dateimanager starten, indem Sie den Delphi-Ordner suchen, öffnen und die Exe-Datei «Delphi32» im Ordner «Bin» doppelklicken, ebenso wie Sie es unter Windows 3.1 getan hätten.

Wir werden Delphi nicht auf diese umständliche Art starten, aber den Explorer benutzen, um uns einen kurzen Einblick in Ordner- und Dateistruktur der Entwicklungsumgebung zu verschaffen.



Starten Sie den WINDOWS-EXPLORER und suchen Sie den Ordner, der die Delphi-Entwicklungsumgebung enthält. Falls Sie nicht während der Installation ein anderes Verzeichnis eingegeben haben, ist dies «Delphi20». Lokalisieren Sie im Ordner «Bin» die Anwendung «Delphi32».

Verschaffen Sie sich anschliessend einen kurzen Überblick über die anderen zur Entwicklungsumgebung gehörigen Ordner, z.B. «RptSmith» oder «Database Desktop», und versuchen Sie anhand der Ordner- und Dateinamen herauszufinden, welche Zusätze und Werkzeuge darin enthalten sein könnten.



Die normale Art, unter Windows 95 ein Programm zu öffnen, bedient sich aber der Schaltfläche START, die unten links mit der Task-Leiste eingeblendet wird, sobald Sie mit der Maus an den unteren Bildschirmrand fahren. Klicken Sie mit der Maus auf START und fahren Sie auf dem erscheinenden Klappmenü auf PROGRAMME. Falls die Installation erfolgreich verlaufen ist, befindet sich nun unter den angezeigten Ordnern «Delphi 2.0».



Delphi 2.0

Bevor Sie das Programm öffnen, klicken Sie auf den Ordner «Borland Delphi 2.0», der ungefähr einer Programmgruppe unter Windows 3.1 entspricht. Der Inhalt sollte etwa wie in Abb. 2 aussehen. Sie finden sicher eine Verknüpfung mit der Entwicklungsumgebung selbst.

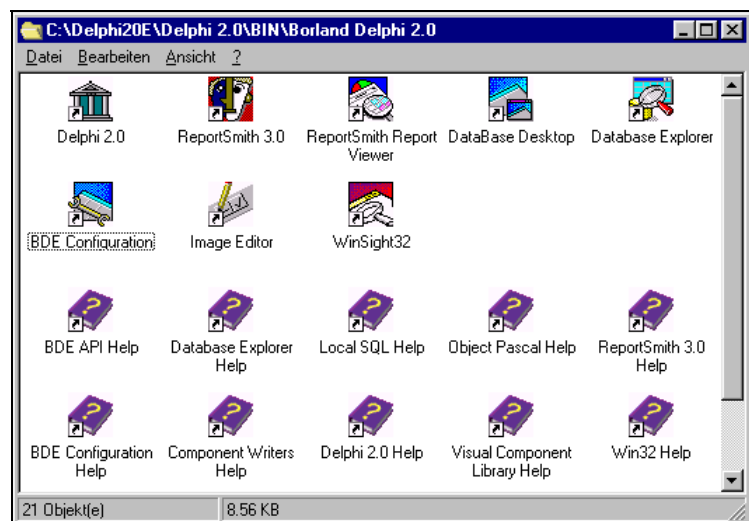


Abb. 2 Inhalt des Ordners «Borland Delphi 2.0»

Je nach Version und installierten Werkzeugen erscheinen daneben der «Report Smith», der «Image Editor» und di-

verse Datenbanktools. Ausserdem sind die verschiedenen Teile der Online-Hilfe aufgelistet. Dank ihres systematischen Aufbaus und ihrer Ausführlichkeit ist die Online-Hilfe ein unentbehrliches Arbeitsinstrument. Sie sollten sich deshalb mit ihrer Handhabung so bald wie möglich vertraut machen.

Nach diesem kurzen Überblick von aussen werden wir nun die Entwicklungsumgebung laden, um sie von innen kennenzulernen.



Die Entwicklungsumgebung lässt sich öffnen, indem man entweder das Icon «Delphi 2.0» im noch geöffneten Ordner doppelklickt, oder indem man das gleichnamige Icon unter Start in der Task-Leiste anklickt. Während des Ladens wird die Entwicklungsumgebung gleichzeitig in die Task-Leiste aufgenommen.

## 2.2 Die Delphi-Oberfläche

Delphi besteht aus der Entwicklungsumgebung und einigen externen Tools. Die Entwicklungsumgebung heisst in Englisch **Integrated Development Environment** oder abgekürzt **IDE**.



Nach dem Start der Entwicklungsumgebung sieht man wie in Abb. 3 jene Elemente der Delphi-Oberfläche, die man für die Erstellung von Windows-Applikationen am häufigsten benötigt.

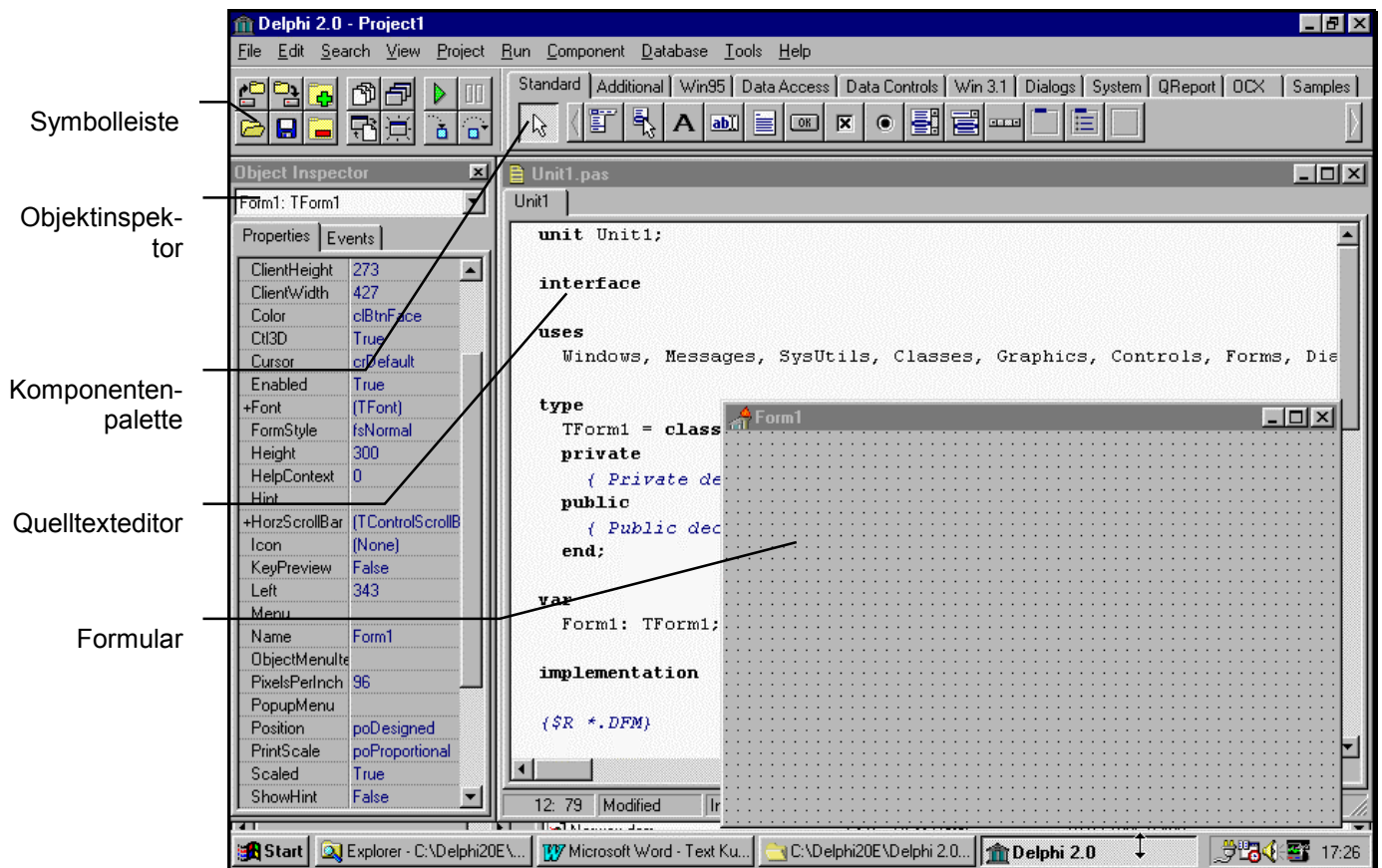


Abb. 3 Die Entwicklungsumgebung von Delphi

Es handelt sich um die folgenden fünf Elemente:





- ☞ • **Symbolleiste** oder in Englisch speedbar
- ☞ • **Komponentenpalette** bzw. component palette
- ☞ • **Objektinspektor** oder object inspector
- ☞ • **Formular** bzw. form
- ☞ • **Quelltexteditor** oder code editor

Wer von Visual Basic umsteigt, wird Teile der IDE wiedererkennen und sich in Delphi leicht zurechtfinden. Doch auch wer mit visueller Programmierung nicht vertraut ist, wird sich an diese grundlegenden Werkzeuge rasch gewöhnen.

Zu den beim Start angezeigten Elementen kommen fünf weitere, die man bei Bedarf über das Menü oder im Fall des Menü-Designers über die Komponentenpalette holen kann, nämlich

- ☞ • **Projektverwaltung** (project manager),



-  • **Menü-Designer,**
-  • **Integrierter Debugger,**
-  • **Objekt-Browser** und
-  • **Bildeditor.**

Projektverwaltung und Debugger werden Sie wie die sichtbaren Elemente bald benutzen. Den Menü-Designer lernen Sie in einer Übung in der zweiten Hälfte des Kurses kennen. In den Bildeditor und den Objekt-Browser führt Sie dagegen dieser Kurs nicht ein. Mit der Online-Hilfe und etwas Experimentierfreude können Sie diese Instrumente selbst erproben.

#### 2.2.1 Wichtige Einstellungen der IDE

Bevor Sie damit beginnen, kleine Beispielprogramme zu erstellen, ist es unbedingt erforderlich, folgende Einstellung der Entwicklungsumgebung vorzunehmen.



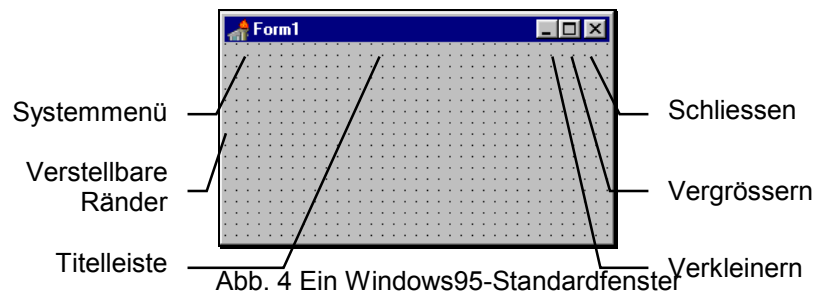
Wählen Sie das Register PREFERENCES im Menü TOOLS - OPTIONS und markieren Sie die beiden Einträge unter AUTOSAVE OPTIONS.

Ausserdem empfiehlt es sich, auch SHOW COMPILER PROGRESS zu aktivieren, um einmal den schnellsten Compiler der Welt bei der Arbeit zu beobachten.

### 2.3 Sichtbare Elemente der Entwicklungsumgebung

#### 2.3.1 Das Formular

Ein Formular, wie Sie es in Abb. 4 sehen, ist der Grundbaustein jeder Delphi-Applikation unter Windows. Es bietet eine leere Fläche, die Sie mit Hilfe vorgefertigter Komponenten zur Benutzerschnittstelle ihres Programms gestalten werden. Auf dem Formular platzieren Sie Menüs, Knöpfe, Eingabefelder und alles, was es dazu sonst noch an visuellen Elementen braucht, um mit der Benutzerin zu kommunizieren.



Ein Formular enthält ohne eine Zeile Programmcode bereits die Grundfunktionalität eines Windows-Fensters, nämlich

- **Titelleiste,**
- **Systemsteuerung,**
- Ränder, mit denen die Grösse geändert werden kann,
- Schaltflächen, um das Formular zu verkleinern, vergrössern oder schliessen.



Dies probieren Sie gleich aus: Beim Öffnen der Entwicklungsumgebung hat Delphi für Sie unter dem Namen «Project1» bereits das Grundgerüst für ein neues Projekt eingerichtet und ein Formular «Form1» angelegt. Wählen Sie jetzt RUN im Menü RUN oder drücken Sie F9, damit dieses Projekt kompiliert und ausgeführt wird. Mit diesem Tastendruck haben Sie Ihr erstes Delphi-Programm erstellt. Es ist zwar noch absolut nutzlos, aber es lässt sich bereits wie jedes Programm unter Windows 95 verkleinern, vergrössern, auf dem Bildschirm herumschieben und schliessen. Testen Sie es! Um mit der Programmierung weiterzufahren, müssen Sie Ihre Anwendung auf jeden Fall schliessen. Der Objektinspektor steht zur Laufzeit nämlich nicht zur Verfügung.

Mit dieser Übung haben Sie auch bereits mit dem integrierten Debugger Bekanntschaft gemacht. Es ist Ihnen vermutlich gar nicht aufgefallen, denn wie der Name es schon sagt ist der Debugger nahtlos in die Entwicklungsumgebung integriert, so dass Sie ihn beim Kompilieren fehlerfreier Programme gar nicht bemerken. Bei der Fehlersuche werden Sie seine Dienste, u.a. Syntaxüberwachung und Haltepunkte, bald schätzen lernen.

### 2.3.2 Der Objektinspektor

Wenn Sie an ihrem Programm weiterarbeiten möchten, dann benötigen Sie dazu als nächstes den Objektinspektor. Mit diesem Element lassen sich Darstellung und Eigenschaften von Formularen und darauf enthaltenen Komponenten während des Entwurfs anpassen. Eigenschaften, die

nur zur Laufzeit existieren, zeigt der Objektinspektor nicht an. Wenn Sie sich in der Online-Hilfe die Liste der Properties einer Komponente zeigen lassen, dann sind diese Laufzeit-Eigenschaften mit einem kleinen grünen Dreieck gekennzeichnet.

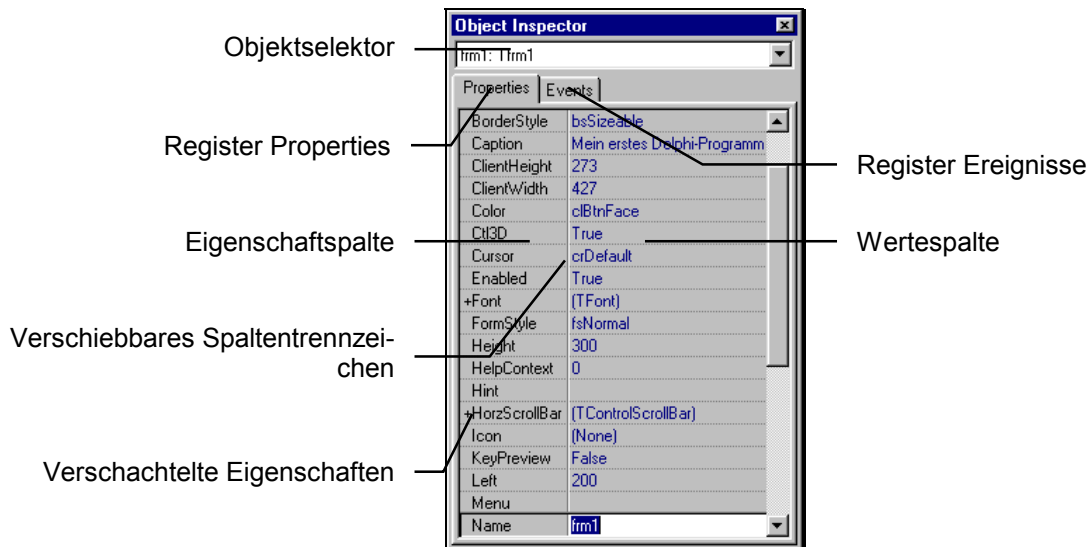


Abb. 5 Der Objektinspektor

☞ Der **Objektselektor** unterhalb des Titelfalkens zeigt Namen und Typ des ausgewählten Objekts. Wenn Sie auf den Pfeil rechts klicken, erscheint eine Liste aller zum Formular gehörigen Komponenten. Zur Zeit befindet sich hier nur `Form1`, weil Sie noch keine Komponenten hinzugefügt haben.

Die zwei Register **PROPERTIES** (Eigenschaften) und **EVENTS** (Ereignisse) sind Schnittstellen zu den Eigenschaften und Ereignisbehandlungsroutinen der Komponenten.

☞ Suchen Sie im Register **PROPERTIES** die Eigenschaften **CAPTION** und **NAME**. Im Feld rechts werden Sie bei beiden `Form1` finden. Als nächstes geben Sie Ihrem Formular eine neue Beschriftung. Tragen Sie unter **NAME** nun `frm` ein. Wie Sie sehen, hat sich auch der Eintrag unter **CAPTION** geändert.

Es ist äusserst wichtig, den Unterschied zwischen **CAPTION** und **NAME** zu verstehen. Unter **NAME** trägt man den **Bezeichner** ein, mit dem eine Komponente im Programm identifiziert wird. Bezeichner unterliegen den folgenden verbindlichen Object-Pascal-Regeln:

- Bezeichner können eine beliebige Länge haben, signifikant sind aber nur die ersten 63 Zeichen.

- Das erste Zeichen eines Bezeichners muss ein Buchstabe oder der Unterstrich ( \_ ) sein.
- Die übrigen Zeichen müssen Buchstaben, Zahlen oder Unterstriche sein.
- Leerzeichen und Sonderzeichen sind innerhalb von Bezeichnern nicht zugelassen.
- Gross- und Kleinschreibung spielt keine Rolle.

CAPTION ist dagegen die Beschriftung einer Komponente, z.B. einer Taste oder eines Formulars und kann ein beliebiger String sein. Alle visuellen und nichtvisuellen Komponenten weisen einen Namen auf, ein CAPTION haben dagegen nur gewisse visuelle Komponenten. Im Skript und insbesondere in den Übungen bezieht sich die Aufforderung, eine Komponente zu **benennen**, auf die Eigenschaft NAME , während **beschriften** im Zusammenhang mit der Eigenschaft CAPTION verwendet wird.

Defaultmässig entspricht CAPTION dem NAME und macht dessen Änderungen mit, solange Sie Caption nicht ändern. Genau dies werden wir jetzt tun.



Geben Sie unter CAPTION einen beliebigen Titel, z.B. „*Mein erstes Delphi-Programm*“ ein und ändern Sie dann Name in *frmprog1*. Mit der ersten Änderung wechselt die Beschriftung im Titelbalken. Von der zweiten Änderung bleibt die Eigenschaft CAPTION dagegen unberührt.



Gewisse Eigenschaften, z.B. FONT, haben selbst wieder Eigenschaften, sogenannte **verschachtelte Eigenschaften**. Im Objektinspektor erkennt man sie am vorangestellten Pluszeichen. Wenn man dieses doppelklickt, öffnet sich eine Liste mit Eigenschaften wie COLOR, HEIGHT und NAME.



Geht man in die Wertespalte der Eigenschaft Font, dann erscheint rechts eine Schaltfläche mit drei Punkten. Ein Mausklick darauf öffnet ein Dialogfenster, in dem die verschachtelten Eigenschaften von FONT gesetzt werden können.



Um sich mit der Änderung der Eigenschaften im Objektinspektor vertraut zu machen, können Sie beispielsweise noch die Farbe des Formulars ändern. Klicken Sie in das Eingabefeld rechts von COLOR, so dass sich ein Listenfeld mit verschiedenen Farboptionen öffnet. Im unteren Teil finden Sie Farben der Windows-Umgebung, im oberen reine Farben. Suchen Sie *clyellow*, um das Formular mit einem sonnigen Gelb einzufärben.

### 2.3.3 Die Komponentenpalette

**Komponenten** sind ein entscheidender Schritt beim Übergang von der herkömmlichen Software-Entwicklung zur Software-Montage. Komponenten können sichtbare Bestandteile von Programmen sein, z.B. Schaltflächen, aber auch unsichtbare, beispielsweise Timer. Statt selbst die ganze Benutzerschnittstelle zu programmieren, greift man teilweise auf vorgefertigte Komponenten zurück. Die Vorteile davon sind mannigfaltig: Kosten- und Zeitbedarf für die Entwicklung reduzieren sich, es treten weniger Fehler auf und Aussehen und Handhabung der Programme werden vereinheitlicht.

Um das nackte Formular mit Komponenten einzukleiden, greifen Sie zur Komponentenpalette.



Abb. 6 Die Komponentenpalette

Funktional zusammengehörige Komponenten sind in verschiedenen Registern der Komponentenpalette zusammengefasst.



Ein Klick auf das Register DIALOGS fördert beispielsweise Windows-Standarddialoge zutage, mit denen sich u.a. Dateien öffnen, drucken und speichern lassen. Um sich Hilfe zu den einzelnen Komponenten der Palette anzeigen zu lassen, klicken Sie mit der rechten Maustaste in die Palette und wählen im Kontextmenü HELP. Die erscheinende Hilfeseite gibt Ihnen Zugang zur Hilfe für die verschiedenen Register und Komponenten. Probieren Sie es aus, indem Sie sich im Register STANDARD die Hilfe zu der Schaltfeldkomponente holen. Mit **Schalter**, **Schaltfläche** oder **Schaltfeld** bezeichnet Delphi übrigens das, was man auf gut Neudeutsch einen Button nennt.



Die Komponentenpalette enthält nicht nur zahlreiche vorgefertigte Komponenten, sondern sie lässt sich auch mit eigenen und zugekauften Komponenten erweitern. Sie können die Palette ganz nach Ihren eigenen Bedürfnissen konfigurieren. Das Benutzerhandbuch hilft dabei weiter.

Als nächstes erweitern Sie nun Ihr Formular mit einem Schaltfeld. Klicken Sie dazu auf die Komponente Schaltfeld und anschließend in ihr Formular `frmprog1`, um das Schaltfeld zu platzieren.



Ein Doppelklick auf die gewünschte Schaltfläche der Komponentenpalette hätte das Schaltfeld genau in der Mitte des Formulars abgesetzt. Da die Komponente zur Zeit ausgewählt ist, sollte sie wie in Abb. 7 von acht Quadraten umrahmt sein, mit deren Hilfe sich die Grösse verändern lässt.

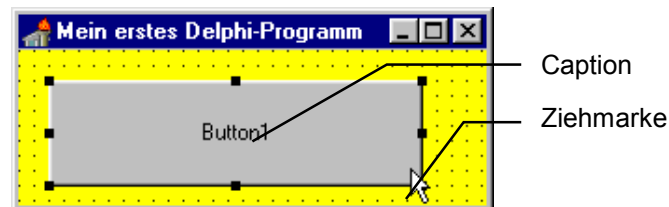


Abb. 7 Ändern der Komponentengrösse

Ziehen Sie an der rechten unteren Ecke, um den Button zu vergrössern. Um ihn in der Mitte des Formulars zu zentrieren, benötigen Sie die ALIGNMENT PALETTE (Ausrichtungspalette), die Sie im Menü VIEW finden. Mit den zwei Symbolen in der mittleren Kolonne lässt sich die Schaltfläche zentrieren.



Abb. 8 Ausrichtungspalette



Vielleicht haben Sie schon bemerkt, dass im Objektinspektor ein neues Objekt namens `Button1` vom Typ `TButton` erschienen ist. Die von der IDE vergebenen Namen sollte man so schnell wie möglich in aussagekräftigere Bezeichnungen umwandeln. Deshalb geben Sie nun rechts von NAME gemäss den eingeführten Namenskonventionen `btnFarbwechsel` ein. Unter Caption kommt „Formularfarbe wechseln“.

Mit dieser Beschriftung ist bereits angedeutet, welche Funktion die Schaltfläche haben soll. Damit Sie diese auch umsetzen können, benötigen Sie das vierte sichtbare Element, nämlich den Quelltexteditor.

#### 2.3.4 Der Quelltexteditor

Mit dem **Quelltexteditor** haben Sie Zugriff auf den gesamten Quellcode eines Projektes.

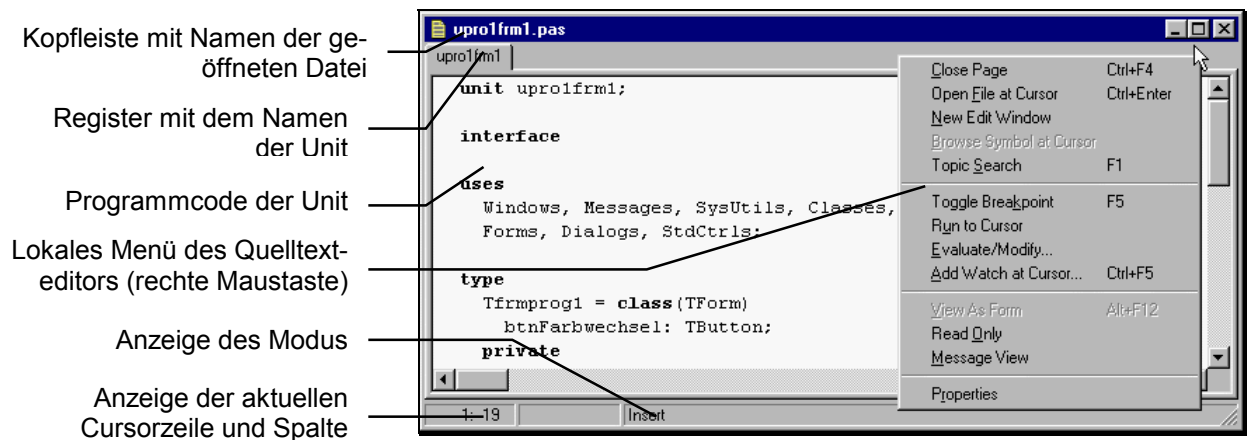


Abb. 9 Elemente des Quelltexteditors

Wie bei allen Elementen der Entwicklungsumgebung finden Sie zusätzliche Information in der Hilfe.



Wählen Sie **HELP - HELP TOPICS**. In der angezeigten Liste öffnen Sie mit einem Doppelklick das Buch „Using Delphi“ und darin den Titel „Programming Environment“. Sie erhalten eine Liste sämtlicher Elemente der Entwicklungsumgebung. Öffnen Sie „Code editor“ und doppelklicken Sie das Thema „About the Code editor“. Lesen Sie den Text durch.



Den Umgang mit der Online-Hilfe sollten Sie sich bald zur Routine machen. Sowohl für den Umgang mit der Entwicklungsumgebung wie für die Sprachelemente von Object Pascal ist sie die beste Quelle, ja in manchen Fällen sogar die einzige, beispielsweise für Änderungen von einer Version zur nächsten.



Sie werden nun Programmcode schreiben. Klicken Sie dafür zuerst auf ihre Schaltfläche `btnFarbwechsel`. Damit wird dieses Objekt im Objektinspektor aktiviert. Wenn Sie ins Register **EVENTS** wechseln, erhalten Sie eine Liste von Ereignissen. Für jedes davon könnten Sie jetzt Routinen schreiben, mit denen das Programm auf dieses Ereignis reagieren soll. „**Routine**“ ist übrigens ein Sammelbegriff für Prozeduren, Funktionen und Methoden. Windows-Programme sind ereignisgesteuert. Die Entwicklung von Windows-Applikationen unter Delphi besteht deshalb zu einem grossen Teil darin, **Ereignisbehandlungsroutinen** zu programmieren.



Vorerst wollen wir nur eine Routine schreiben, die auf einen Mausklick reagiert. Das Ereignis dazu ist das oberste in der Liste mit Namen **ONCLICK**. Wenn Sie in das Feld daneben doppelklicken, passiert erstaunliches: Delphi wechselt von selbst in den Quelltexteditor und erstellt den Kopf der Prozedur zur Behandlung

dieses Ereignisses. Sie finden auch bereits `begin` und `end` vor, so dass Sie in der Zeile dazwischen nur noch eingeben müssen:

```
frmprog1.color := clred;
```

Mit dieser Zeile weisen Sie der Eigenschaft `color` des Formulars `frmprog1` neu die Farbe `clred`, d.h. Rot, zu. Abgeschlossen wird jede Anweisung in Object Pascal mit einem **Strichpunkt**. Sie sollten jetzt also den folgenden Code vor sich haben:

```
1 procedure Tfrmprog1.btnFarbwechselClick
2   (Sender: TObject);
3 begin
4   frmprog1.color := clred;
5 end;
```



Klicken Sie in der Symbolleiste auf den grünen Pfeil RUN, um das Projekt zu kompilieren und anschliessend laufen zu lassen. Wenn Sie alles richtig gemacht haben, „errötet“ ihr Formular, sobald Sie auf die Schaltfläche klicken.

### 2.3.5 Die Symbolleiste

Die Handhabung der Symbolleiste dürfte Ihnen aus anderen Windows-Anwendungen bekannt sein, so dass auf sie hier nicht speziell eingegangen wird. Wenn Sie mit der Maus auf die einzelnen Symbole zeigen, erhalten Sie Schnellhinweise über deren Funktion.

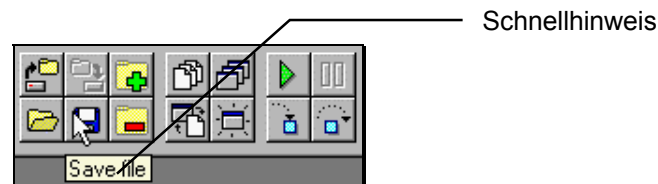


Abb. 10 Symbolleiste



Benutzen Sie jetzt die Schaltfläche SAVE FILE, um Ihr Projekt zu speichern. Als erstes erhalten Sie die Dialogmaske „Save Unit1 As“. Unter Dateiname ist «Unit1» eingetragen. Weil es sich um eine Unit handelt, die zum ersten Formular Ihres ersten Projektes gehört, tragen Sie als Dateiname nun beispielsweise *upro1frm1* ein. Da wir uns unter Windows 95 befinden, können Sie selbstverständlich auch einen längeren Namen eingeben. Die Eingabe dient allerdings nicht nur als Dateiname, sondern auch als Name der Unit im Delphi-Code. Deshalb unterliegt dieser Name den Object-Pascal-Konventionen für Bezeichner (siehe Abschnitt 2.3.2) und darf im Gegensatz zu den langen Dateinamen keine Leerschläge und Sonderzeichen enthalten. Weil diese Eingabe gleichzeitig auch beim Objektinspektor im Tabulator des Registers ihrer Unit erscheint, sollten Sie sie kurz halten, damit Sie



auch bei Projekten mit mehreren Units noch den Überblick behalten.



Gewöhnen Sie sich auch an, für **jedes Projekt** einen **neuen Ordner** anzulegen. Ausserdem ist es sicherer, die eigenen Dateien von den Programmen zu trennen, was Windows und andere Microsoft-Programme nun im Gegensatz zu Delphi unterstützen und automatisch vorschlagen. Suchen Sie mit Hilfe des Listenfeldes den Ordner «Eigene Dateien» oder einen anderen Ordner mit Benutzerdateien. Erstellen Sie darin mit Hilfe des Schaltfeldes einen neuen Ordner «Delphi» und in diesem wiederum einen Ordner «Prog1».



Leider haben Sie damit noch nicht alles gespeichert. Ein **Projekt** gliedert sich nämlich in eine **Projektdatei** und mehrere **Units**. Wir gehen im Unterkapitel „Die Dateien eines Projekts“ noch genauer darauf ein. Wenn Sie die Entwicklungsumgebung verlassen, werden Sie automatisch gefragt, ob Sie die Projektdatei speichern möchten. Sicherer ist es aber, dies schon vorher zu tun, indem man das Menü benutzt.



Wählen Sie SAVE PROJECT AS im Menü FILE, und speichern Sie Ihre Projektdatei unter *PProg1*.

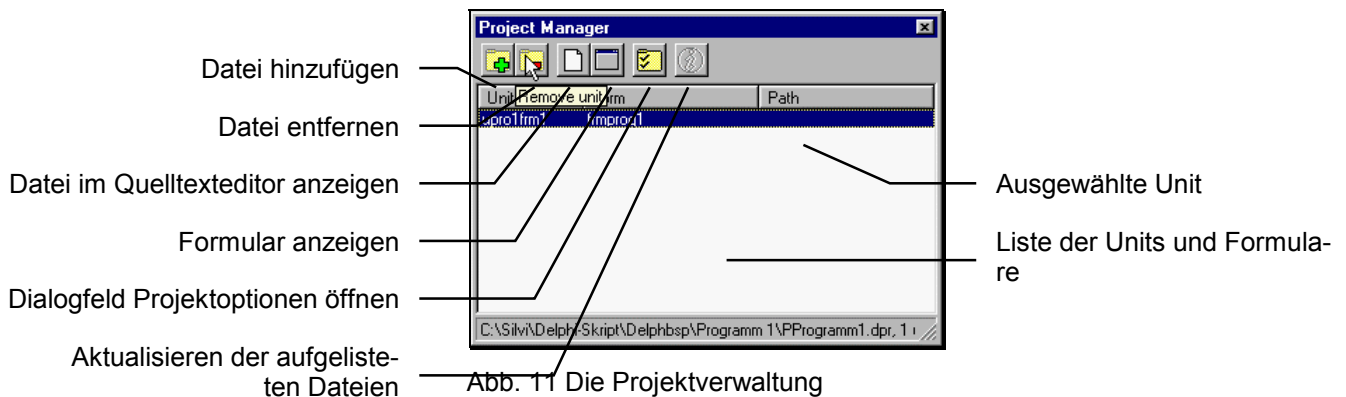
## 2.4 Die Projektverwaltung

Delphi ist eine Sprache, welche die Wiederverwendung von Code stark unterstützt. Ein wichtiger Schritt in diese Richtung ist, dass sich ein Projekt aus verschiedenen **Units** zusammensetzt. Units sind Sammlungen von Konstanten, Typen, Variablen, Prozeduren und Funktionen, die von mehreren Anwendungen gemeinsam verwendet werden können. Das Kapitel 3 geht später ausführlich darauf ein.

Wenn Sie sich einen Überblick verschaffen möchten, welche Units zu einem Projekt gehören, dann hilft die Projektverwaltung weiter.



Öffnen Sie im Menü VIEW das Untermenü PROJECT MANAGER. Ein Dialogfenster wie in Abb. 11 öffnet sich. Klicken Sie auf die Zeile, die mit *upro1frm1* beginnt.



In der ersten Spalte sehen Sie eine Liste der vorhandenen Units und in der zweiten Spalte die damit verknüpften Formulare. Jedes Formular hat eine eigene Unit. Da es auch Units ohne Formulare gibt, kann diese Spalte leer sein. Die Spalte „Path“ ist meistens leer, denn der Pfad wird nur für jene Units angegeben, die in einem anderen Ordner gespeichert sind als die Projektdatei.

Doch die Dateiverwaltung dient nicht nur zur Anzeige. Mit Hilfe der Schaltflächen am oberen Rand können Sie auch Units zum aktuellen Projekt hinzufügen oder vom Projekt entfernen, zwischen Units und Formularen wechseln sowie die Optionen eines Projekts ändern. Mit der rechten Maustaste erhalten Sie wie bei Quelltexteditor Zugang zu einem lokalen Menü.



Zur Zeit existiert für Ihr Projekt nur das Formular `frmprog1` in der Unit «`upro1frm1`», die in der gleichnamigen Datei mit der Endung «`.pas`» gespeichert ist. Zum Formular wechseln Sie, indem Sie `frmprog1` doppelklicken. Klicken Sie die Projektverwaltung an, um sie erneut zu aktivieren, und schliessen Sie sie.

## 2.5 Die Dateien eines Projekts

Delphi ist ein echter und extrem schneller Compiler. Die ausführbaren EXE- und DLL-Dateien enthalten Maschinencode, der vom Prozessor direkt ausgeführt werden kann. Deshalb sind die mit Delphi erzeugten Applikationen vor allem gegenüber interpretierten Sprachen beträchtlich schneller. Doch die weitere Optimierung des Compilers bei der Version 2.0 führt dazu, dass Delphi-Applikationen auch schneller sind als andere kompilierte Programme, beispielsweise von C++.

Grundsätzlich unterscheidet man drei Kategorien von Dateien, die bei Delphi-Projekten vorkommen:

- beim Entwurf erstellte Dateien
- vom Compiler erstellte Projektdaten
- nicht von Delphi erstellte Ressourcendateien (Bitmaps, Icons und Hilfe-Dateien)

Die folgende Tabelle aus der Online-Hilfe gibt einen Überblick über die verschiedenen Dateierweiterungen in der ersten Kategorie:

<b>.DPR Projektdaten</b>	Object Pascal-Quellcode der Hauptdatei des Projekts. Listet alle Formulare und Units des Projekts auf und enthält Code für die Initialisierung der Anwendung. Wird beim ersten Speichern des Projekts erstellt.
<b>.PAS Unit-Quellcode</b>	Für jedes Formular, das im Projekt enthalten ist, wird eine .PAS-Datei erstellt, wenn das Projekt zum ersten Mal gespeichert wird. (Selbstverständlich kann Ihr Projekt auch .PAS-Dateien enthalten, die nicht mit Formularen verbunden sind.) Enthält alle Deklarationen und Prozeduren einschließlich der Event-Handler des Formulars.
<b>.DFM Graphische Formulardatei</b>	Binärdatei, Textdatei in Delphi, welche die Entwurfsmerkmale eines im Projekt enthaltenen Formulars enthält. Für jedes Formular wird zusammen mit der entsprechenden .PAS-Datei eine .DFM-Datei beim ersten Speichern des Projekts erstellt.
<b>.DOF Projektoptionendatei (Delphi 1.0: .OPT)</b>	Textdatei, welche die aktuellen Projektoptionen enthält. Wird beim ersten Speichern angelegt und bei allen Änderungen der Projektoptionen aktualisiert.
<b>.RES Compiler-Ressourcen-Datei</b>	Binärdatei, die das Anwendungs-Icon und andere im Projekt verwendete externe Ressourcen enthält.
<b>.DSK Desktop-Einstellungen</b>	In dieser Datei werden die Desktop-Einstellungen gespeichert, die Sie für das Projekt im Dialogfenster Umgebungsoptionen angegeben haben.
<b>.~* Backup-Dateien</b>	Kopie der DPR-, PAS- und DFM-Dateien, wie sie vor dem letzten Speichern existierten

Vom Compiler erstellte Dateien gibt es die folgenden:

<b>.EXE</b> <b>Kompilierte, ausführbare Datei</b>	<b>aus-</b>	Dies ist die vertreibbare, ausführbare Datei Ihrer Anwendung. In sie werden alle notwendigen .DCU-Dateien beim Kompilieren Ihrer Anwendung aufgenommen. Ihre Anwendung kann also ohne weitere .DCU-Dateien vertrieben werden.
<b>.DCU</b> <b>Unit-Objekt-Code</b>		Beim Kompilieren wird für jede .PAS-Datei Ihres Projekts eine .DCU-Datei erzeugt.
<b>.DLL</b> <b>Kompilierte dynamische Link-Bibliothek</b>		Information über die Erzeugung von DLLs entnehmen Sie bitte dem Delphi-Komponentenhandbuch.

Dazu kommen noch die nicht von Delphi erstellten Dateien:

<b>.BMP</b> <b>Bilddateien</b>		Die Bitmaps (.BMP-, .WMF-Dateien) können Sie in Komponenten vom Typ <code>TImage</code> oder <code>TBitBtn</code> verwenden. Sie werden in die ausführbare Datei des Projekts aufgenommen.
<b>.WMF</b>		
<b>.ICO</b> <b>Icons</b>		Icons können Sie im Dialogfenster Projektoptionen oder für ein Formular als Merkmal angeben. Wie Bitmap-Dateien werden sie in das Projekt aufgenommen. BMP- und ICO-Dateien lassen sich mit dem Bildeditor von Delphi erstellen.
<b>.HLP</b> <b>Windows-Hilfdateien</b>		Im Dialogfenster Projektoptionen können Sie für ein Projekt eine Online-Hilfdatei angeben. Die .HLP-Datei kann mit dem mit Delphi ausgelieferten Microsoft Hilfe-Compiler oder einem Hilfekompiler eines anderen Drittanbieters erstellt werden.



Sehen Sie sich die Dateien Ihres Projektes an. Wechseln Sie dazu in den Explorer, und suchen Sie den Ordner mit ihren Projektdateien. Falls die Endungen nicht angezeigt werden, können Sie dies ändern, indem Sie im Menü **ANSICHT** auf **Optionen** klicken, das Register **ANSICHT** wählen und das Kreuzchen bei „Keine MS-DOS-Erweiterungen“ entfernen.

Fasst man die Beziehung der wichtigsten Dateien eines Projektes in einer Grafik zusammen, dann ergibt sich das Bild in Abb. 12:

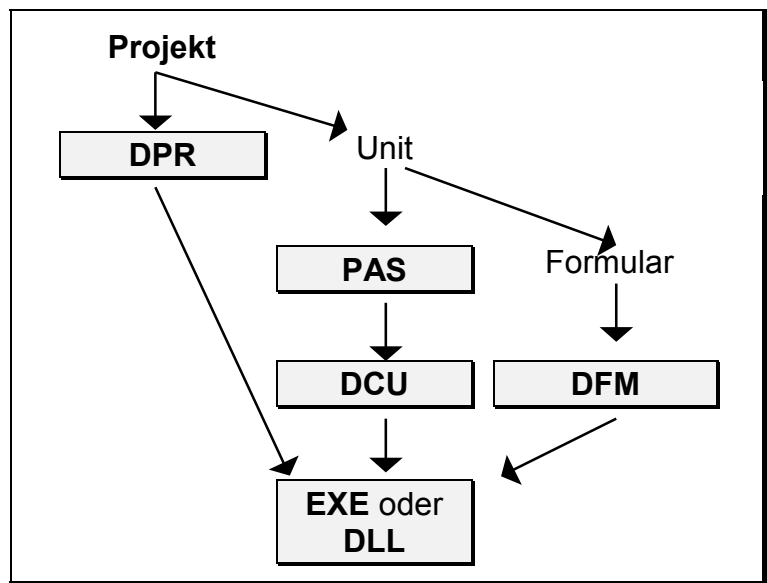


Abb. 12 Die wichtigsten Dateien eines Projektes

## 2.6 Der integrierte Debugger

Gleichgültig, wie sorgfältig Sie beim Schreiben des Quelltextes vorgehen, Ihre Programme werden vermutlich Fehler (bugs) enthalten, die verhindern, dass die Programme so ausgeführt werden, wie Sie es wünschen. Den Prozess des Aufspürens und Behebens dieser Fehler nennt man **Debugging**.

Es gibt drei Arten von Fehlern:

- Syntaxfehler
- Laufzeitfehler / Logische Fehler

Delphi bietet Funktionen zur Fehlerbeseitigung, die im integrierten Debugger zusammengefasst sind und die Sie bei der Suche nach Fehlern in Ihrem Programm und deren Behebung unterstützen.

Der integrierte Debugger versetzt Sie in die Lage,

- die Ausführung Ihres Programms zu steuern,
- Werte von Variablen und Elementen in Datenstrukturen zu überwachen
- und Werte von Datenelementen während einer Ausführung unter Kontrolle des Debuggers zu modifizieren.

Vor dem Gebrauch des Debuggers müssen Sie Ihr Programm mit symbolischen Debug-Informationen kompilieren.

Danach können Sie eine Debug-Sitzung beginnen, indem Sie Ihr Programm unter Delphi ausführen. Der integrierte Debugger übernimmt jedesmal die Kontrolle, wenn Sie Ihren Programmcode ausführen, überprüfen oder schrittweise abarbeiten lassen.

Das Programm verhält sich bei der Ausführung unter der Kontrolle des Debuggers genauso wie sonst auch. Es erzeugt Fenster, nimmt Benutzereingaben an, berechnet Werte und zeigt Resultate an. Wenn das Programm gerade nicht ausgeführt wird, liegt die Kontrolle beim Debugger, und Sie können dessen Funktionen benutzen, um den aktuellen Zustand des Programms zu untersuchen. Indem Sie die Werte von Variablen, die Hierarchie der aufgerufenen Funktionen und die Programmausgabe inspizieren, können Sie sicherstellen, dass der untersuchte Bereich des Programmcodes wie beabsichtigt arbeitet. Sie finden diese Instrumente unter den folgenden Menüpunkten:

<b>Instrument</b>	<b>Menü</b>
Haltepunkt (breakpoint) setzen	RUN - ADD BREAKPOINT oder Mausklick am linken Rand der Zeile im Quelltexteditor -> Zeile wird rot unterlegt
Liste der Haltepunkte ansehen	VIEW - BREAKPOINTS
Ausdruck überwachen (watch)	RUN - ADD WATCH - Bezeichner oder CTRL + F5
Liste der überwachten Ausdrücke ansehen	VIEW - WATCHES
Aufrufhierarchie ansehen	VIEW - CALL STACK

## 2.7 Rekapitulation

### 2.7.1 Kapiteltest

- 1) Was heisst IDE?
- 2) Zählen Sie die zehn verschiedenen Elemente der Entwicklungsumgebung auf, beschreiben Sie ihre Funktion und teilen Sie sie in die zwei Kategorien „beim Start sichtbar“ und „beim Start nicht sichtbar“ ein.
- 3) Welche der folgenden Bezeichner sind gültig:

Max' Verkaeufe

Dies\_ist\_der\_laengste\_Bezeichnung\_den\_ich\_mir  
\_in\_einem\_Delhiprogramm\_vorstellen\_kann (Ein  
Wort!)

\_1994Absatz

1994Absatz

Absatz\_1994

x


- 4) Geben Sie für die folgenden Aussagen an, ob sie richtig oder falsch sind:
- a) Die Hauptdatei eines Projektes hat die Endung «.dfm».
  - b) Der Objektinspektor zeigt alle Eigenschaften einer ausgewählten Komponente.
  - c) Ein Projekt hat mindestens so viele Units, wie es Formulare hat.

#### 2.7.2 Übung



Generieren Sie ein Projekt PProg1\_2, in dem Sie mit Hilfe der Label-Komponente aus dem Register Standard das berühmte „Hello world!“ mit grosser, gelber, fatter Schrift in der Mitte eines dunkelblauen Formulars anzeigen. Ändern Sie den Formulartitel und vergeben Sie Formular und Label Namen gemäss den Namenskonventionen aus Kapitel 1.5.3.

### 3 Programme und Units

 Im vorangehenden Kapitel haben Sie bereits gehört, dass ein Projekt sich aus verschiedenen wiederverwendbaren Units, d.h. Sammlungen von Konstanten, Typen, Variablen, Prozeduren und Funktionen, zusammensetzt. In diesem Kapitel werden wir uns die Struktur von Projekten und Units genauer anschauen.


Sie lernen, dass Units zwei Teile enthalten, einen ersten, in dem die Kommunikation mit anderen Units geregelt wird, und einen zweiten, der den Programmcode der Routinen der Unit enthält. Ausserdem schliessen Sie Bekanntschaft damit, wie Units sich gegenseitig aufrufen können.

Sie werden mit Hilfe eines Beispielprogramms folgende Themen bearbeiten:

- Syntax eines Programms
- Syntax einer Unit
- Interface- und Implementationsteil einer Unit
- Zirkuläre und indirekte Referenzen

#### 3.1 Ein Beispielprogramm

Delphi vereinfacht die Programmierung, indem es allgemeinen Code automatisch in ein Projekt einfügt. Auch mit rudimentären Vorkenntnissen kann man deshalb ein Beispielprogramm erstellen, mit dem sich die Teile eines Projektes erläutern lassen.

 Ihre Beispielapplikation soll folgende Charakteristiken aufweisen: Sie werden ein Formular mit einem **Bitmap-Schalter** erzeugen, der nach dem Anzeigen einer Dialogbox die Applikation schliesst. Ein Bitmap-Schalter unterscheidet sich von der in der ersten Übung verwendeten Schaltfläche einzig dadurch, dass er neben der Beschriftung noch eine Grafik, eben ein Bitmap, enthält.



Gehen Sie folgendermassen vor:

1. Öffnen Sie Delphi, falls dies nicht bereits geschehen ist.
2. Solange die Entwicklungsumgebung nicht anders konfiguriert worden ist, hat Delphi beim Öffnen für Sie bereits ein neues Projekt «Project1» mit einem leeren Formular `Form1` und einer Unit «Unit1.pas» erzeugt. Ändern Sie im Register PROPERTIES des Objektinspektors den Namen des Formulars in



*frmCloseApp* und geben Sie unter CAPTION „*Ein Bitmap-Schalter, um Programme zu schliessen*“ ein.



3. Doppelklicken Sie auf den Bitmap-Schalter, der sich ganz links im Register ADDITIONAL der Komponentenpalette befindet. Der Schalter wird damit automatisch in der Mitte des Formulars platziert.
4. Verlängern Sie den Button mit Hilfe der sichtbaren Ziehmarken.
5. Wechseln Sie in den Objektinspektor, nennen Sie den Schalter *btnbitClose* und beschriften Sie ihn mit „*Anwendung schliessen*“.
6. Suchen Sie die Eigenschaft GLYPH. Ein GLYPH ist eine Grafik, die bis zu vier verschiedene Darstellungsformen eines Bitmap-Schalters in einem einzigen Bitmap speichert. Klicken Sie in die Wertespalte neben der Eigenschaft GLYPH. Rechts erscheint wie bei FONT eine Schaltfläche mit drei Punkten. Wenn Sie diese anklicken, erscheint ein Dialogfenster „**Picture Editor**“. Dabei handelt es sich nicht um den bereits erwähnten Bildeditor, mit dem Bitmaps und Icons bearbeitet werden können. Der Picture Editor erlaubt es nur, für bestimmte grafische Komponenten Bitmaps zu laden.
7. Klicken Sie auf LOAD und wechseln Sie ins Windows-Verzeichnis. Dort sollten diverse kleine Bitmaps vorhanden sein. Wählen Sie «Karo.bmp» oder eine beliebige andere Datei und klicken Sie dann auf ÖFFNEN. Beendet wird die Aktion, indem Sie im Picture Editor OK drücken. Wenn Sie erfolgreich waren, dann erscheint auf dem Bitmap-Schalter links von der Beschriftung die ausgewählte Grafik.
8. Wechseln Sie ins Register EVENTS. Doppelklicken Sie in die Wertespalte neben dem Ereignis ONCLICK.
9. Wie Sie bereits wissen, wechselt Delphi in den Quelltexteditor und erstellt für Sie den Kopf einer Prozedur zur Behandlung dieses Ereignisses. Geben Sie die folgenden zwei Zeilen ein:

```
MessageDlg('Applikation schliessen', mtInformation, [mbOK], 0);  
Close;
```

Als nächstes können Sie Ihr Projekt testen. Zuerst speichern Sie aber die Unit mit FILE - SAVE FILE AS unter dem Dateinamen «*uClose.pas*» in einem neuen Ordner «*Prog2\_1*». Anschliessend speichern Sie das Projekt mit FILE-SAVE PROJECT AS unter dem Dateinamen «*pApp2\_1.dpr*». Führen Sie das Programm jetzt aus, indem Sie F9, die Schaltfläche RUN in der Symbolleiste oder das Menü RUN - RUN betätigen.

Wenn Sie bei der laufenden Anwendung auf den Schalter „Anwendung schliessen“ klicken, sollten Sie dasselbe wie in Abb. 13 erhalten.



Abb. 13 Das Programm «pApp2\_1» zur Laufzeit

### 3.2 Die Syntax eines Programms

Vielleicht ist es Ihnen noch nicht aufgefallen, aber bis jetzt haben Sie den Code der Projektdatei gar nicht zu Gesicht bekommen. Delphi hält die DPR-Datei absichtlich versteckt, weil sie automatisch generiert und aktualisiert wird. Eine manuelle Bearbeitung empfiehlt sich nur für Fortgeschrittene.

#### 3.2.1 Die Syntax des Beispielprogramms



Mit VIEW - PROJECT SOURCE holen Sie den Code der Projektdatei in den Quelltexteditor.

Sie sollten folgenden Code sehen:

```

1  program pApp2_1;
2
3  uses
4      Forms,
5      uClose in 'uClose.pas' {frmCloseApp};
6
7  {$R *.RES}
8
9  begin
10     Application.Initialize;
11     Application.CreateForm(TfrmCloseApp,
12         frmCloseApp);
13     Application.Run;
14 end.

```



Die Projektdatei ist gewissermassen das Hauptprogramm. Hier sind die Formulare und Units verzeichnet und hier ist der Initialisierungscode enthalten. Das ist allerdings auch schon alles, denn alles andere wird in Units ausgelagert. Projektdateien sind deshalb immer sehr kurz.

☞ In Delphi sehen Sie selbstverständlich **keine Zeilennummerierung**. Im Skript wurde sie bei längeren Codebeispielen beigelegt, damit Erklärungen direkt auf einzelne Zeilen Bezug nehmen können.

☞ Zeile 1 enthält das reservierte Wort `program`. **Reservierte Wörter** haben innerhalb der Programmiersprache Object Pascal eine feste Bedeutung. Sie können **nicht umdefiniert** werden. In der Online-Hilfe finden Sie unter dem Stichwort „Reserved Words“ eine Liste aller reservierten Wörter.

`Program` teilt dem Compiler mit, dass hier das Hauptprogramm beginnt. Hinter `program` folgt der Name, den Sie der Applikation gegeben haben. Auf Zeile 3 finden Sie `uses`, ein weiteres reserviertes Wort. Damit weiss der Compiler, dass eine Liste von Units folgt, die ins Hauptprogramm eingebunden werden. In Zeile 4 folgt dann `Forms`, eine von Delphi zur Verfügung gestellte Unit. Zeile 5 enthält Ihre eigene Unit `uClose` und gibt an, dass sie in der Datei «uClose.pas» gespeichert ist.

Der Text in den geschweiften Klammern { } wird vom Compiler ignoriert. Mit diesen Klammern oder auch mit (\* und \*) kennzeichnet man einen Kommentar. Delphi gibt hier freundlicherweise das Formular an, das zu einer Unit gehört. Jedes Formular hat nämlich seine eigene Unit.

☞ Zeile 7 enthält einen **Compiler-Befehl**, in englisch „compiler directive“ genannt. Compiler-Befehle sind spezielle Anweisungen für den Compiler, obwohl sie sich wie Kommentare in geschweiften Klammern befinden. Das von einem Buchstaben gefolgte \$-Zeichen, das eventuell noch von einem Dateinamen ergänzt wird, unterscheidet die Direktive von einem gewöhnlichen Kommentar. {`$R *.RES`} gibt an, dass alle Ressourcendateien im Projektverzeichnis mit der Endung `.res` in das Programm aufgenommen werden sollen.

Das reservierte Wort `begin` in Zeile 9 leitet die Anweisungen des Hauptprogramms ein. Das von einem Punkt gefolgte `end` in Zeile 14 ist das Gegenstück zu `begin` und bildet den Schlusspunkt der Applikation. Dazwischen befinden sich drei Anweisungen, mit denen die Applikation initialisiert, erstellt und aufgerufen wird. Erst das Kapitel über Klassen und Instanzen wird dazu weitere Erklärungen liefern.

Im Hauptprogramm sieht man bereits die klassische Zweiteilung, die auch bei den Units vorhanden ist, und eine grund-

legende Charakteristik von Object Pascal darstellt. Jeder Object-Pascal-Block (das Konzept „Block“ finden Sie in Kapitel 7) weist einen optionalen **Deklarationsteil** und einen **Anweisungsteil** auf. Wenn es einen Deklarationsteil gibt, dann steht er vor dem Anweisungsteil. In unserem Beispielprogramm umklammern `begin` und `end` den Anweisungsteil.

Achten Sie auch auf die Verwendung des Strichpunkts, mit dem eine Anweisung abgeschlossen wird. Im Normalfall wird jede Zeile mit einem Strichpunkt abgeschlossen, weil sie genau eine Anweisung enthält. Ausnahmen sehen Sie in den Zeilen 3, 7, 9 und 14. Die Aufzählung der Units nach `uses` ist eine einzige Anweisung, so dass erst nach der letzten Unit ein Strichpunkt folgt. Auf Compiler-Befehle und Kommentare folgt ebenfalls kein Strichpunkt. Die reservierten Wörter `begin` und `end` bilden aus mehreren Anweisungen eine zusammengesetzte Anweisung, die als einzelne Anweisung behandelt wird. Deshalb folgt erst nach `end`, nicht aber nach `begin` ein Strichpunkt. Das letzte `end` eines Programms wird statt mit einem Strichpunkt mit einem Punkt abgeschlossen.

### 3.2.2 Die allgemeine Syntax eines Programms

Aus dem Beispielprogramm lässt nun die folgende generelle Syntax extrahieren:

```

program Name;

uses    <Liste der verwendeten Units>;
const  <Liste von Konstanten>
type   <Liste von benutzerdefinierten Typen>;
var    <Liste von Variablen>;

<Liste von Funktionen und Prozeduren>

begin
    <Anweisungen>
end.

```

Jedes Pascal-Programm fängt mit dem reservierten Wort `program` an. **Klein- und Grossschreibung spielen** übrigens in Pascal **keine Rolle**, es könnte also auch `ProGram` `NAME` in der ersten Zeile stehen.

### 3.2.3 Reihenfolge der Deklarationen

Im Deklarationsteil deklariert man ausser Units auch Konstanten, Typen, Variablen, Prozeduren und Funktionen. Während `unit`, `interface` und `uses` immer in dieser Reihenfolge erscheinen müssen, ist die hier vorgeschlagene Reihenfolge für Deklarationen zwar häufig anzutreffen, aber nicht zwingend. Das Benutzerhandbuch empfiehlt beispielsweise die Reihenfolge Typen, Variablen, Konstanten, Routinen. Sie können auch mehr als einen Deklarationsteil der gleichen Art verwenden.

☞ Allerdings gilt dabei die eiserne Regel, dass Sie nur auf das Bezug nehmen können, was Sie bereits deklariert haben. Wenn Sie also eine Variable `strMeinName` vom Typ `TMeinTyp` definieren, dann müssen Sie vorher `TMeinTyp` bereits deklariert haben.

Auch innerhalb von Routinen ist wieder ein Deklarationsteil für lokale Typen, Variablen und Routinen anzutreffen.

## 3.3 Die Struktur einer Unit

- ☞ **Units** bilden die Grundlage der modularen Programmierung. Mit Units bilden Sie Bibliotheken und unterteilen grosse Programme in logisch gruppierte Module. Der Begriff **Modul** wird meistens synonym zu Unit gebraucht.

### 3.3.1 Die Unit `uClose` des Beispielprogramms

Unser Beispielprogramm hat eine einzige Unit, an der Sie die allgemeine Syntax einer Unit studieren können. Der Programmcode Ihrer Unit sollte folgendermassen aussehen:

```

1  unit uClose;
2
3  interface
4
5  uses
6      Windows, Messages, SysUtils, Classes,
7      Graphics, Controls, Forms, Dialogs,
8      StdCtrls, Buttons;
9
10 type
11     TfrmCloseApp = class(TForm)
12         btnbitClose: TBitBtn;
13         procedure btnbitCloseClick
14             (Sender: TObject);

```

```

15     private
16         { Private declarations }
17     public
18         { Public declarations }
19     end;
20
21     var
22         frmCloseApp: TfrmCloseApp;
23
24     implementation
25
26     {$R *.DFM}
27
28     procedure TfrmCloseApp.btnbitCloseClick
29         (Sender: TObject);
30     begin
31         MessageDlg('Applikation schliessen',
32             mtInformation, [mbOK], 0);
33         Close;
34     end;
35
36 end.

```

### 3.3.2 Die Syntax einer Unit

Vieles im Code einer Unit kennen wir bereits vom Programm her. Es lässt sich folgende generelle Syntax einer Unit extrahieren:

```
unit Name;
```

```
interface
```

```
uses <Liste der verwendeten Units>;
```

```
const <Liste von Konstanten>
```

```
type <Liste von benutzerdefinierten Typen>;
```

```
var <Liste von Variablen>;
```

```
<Liste von Deklarationen exportierter  
Funktionen und Prozeduren>
```

```
implementation
```

```
uses <Liste der lokal verwendeten Units>;
```

```
const <Liste von Konstanten>
```

```
type <Liste von benutzerdefinierten Typen>;
```

```
var <Liste von Variablen>;
```

```
<Implementierung von Funktionen und
```


Prozeduren>

```
[begin
  <Initialisierungs-Anweisungen>
  <Finalization-Anweisungen>]
end.
```


### 3.3.3 Die Teile einer Unit

In der generellen Syntax einer Unit erkennt man fünf Teile:

1. **Unit-Kopf**
2. **Interface-Teil**
3. **Implementationsteil**
4. **Initialisierungsteil**
5. **Finalization-Teil**


 Der **Unit-Kopf** besteht aus dem reservierten Wort `unit` und einem eindeutigen Bezeichner. Dieser Bezeichner erscheint in der Uses-Klausel von anderen Programmen und Units, welche diese Unit benützen.


 Im Beispielprogramm finden Sie den Unit-Kopf in Zeile 1.

 Der **Interface-Teil** einer Unit bestimmt, was für jedes Programm oder jede andere Unit, die diese Unit verwenden, sichtbar und zugänglich ist.

Der Interface-Teil beginnt mit dem reservierten Wort `interface` und endet vor dem reservierten Wort `implementation`. Der Interface-Teil deklariert Konstanten, Typen, Variablen, Prozeduren und Funktionen für **public** (öffentlich). Dadurch können sie von anderen Programmen oder Units benutzt werden.

Von Prozeduren und Funktionen erscheint nur die Kopfzeile im Interface-Teil, denn sie sind im Implementationsteil implementiert.

 Die Zeilen 3 bis 23 im Beispielprogramm bilden den Interface-Teil. Sie finden keine Konstantendeklaration, aber eine Uses-Klausel in den Zeilen 5 bis 8, die Deklaration eines Typs von Zeile 10 bis 19 und eine Variable in Zeile 22.


 Der **Implementationsteil** einer Unit enthält den Programmteil aller öffentlichen Prozeduren und Funktionen, die im Interface-Teil der Unit deklariert wurden. Er deklariert auch Konstanten, Typen, Variablen, Prozeduren und Funktionen, die privat sind. Die genaue Bedeutung der Begriffe `public`


und `private` werden Sie erst nach dem Kapitel Programmstruktur besser verstehen.

Der Implementationsteil einer Unit ist der `private` Teil. In diesem Teil enthaltene Deklarationen können nur in diesem Abschnitt der Unit verwendet werden. Alle Konstanten, Typen, Variablen, Prozeduren und Funktionen, die im Interface-Teil deklariert wurden, sind im Implementationsteil sichtbar.


Implementierungen von Prozeduren und Funktionen, die im Interface-Teil deklariert wurden, können in jeder Abfolge des Implementationsteils definiert werden.


In der Implementierung kann eine `Uses`-Klausel unmittelbar nach dem reservierten Wort `implementation` folgen. Wenn Sie eine `Uses`-Klausel in den Interface-Teil einer Unit einfügen, dann sind die in der `Uses`-Klausel aufgeführten Units für die definierende Unit nicht sichtbar.

 Im Beispielprogramm finden Sie den Implementationsteil ab Zeile 24. In Zeile 28 bis 34 sehen Sie die Implementierung einer **Methode**. Prozeduren und Funktionen werden Methoden genannt, wenn sie zu einer Klasse gehören. Die Deklaration dieser Methode erfolgte innerhalb der Klasse `TfrmCloseApp`, und zwar in Zeile 13 und 14.

 Der optionale **Initialisierungsteil** einer Unit umfasst das reservierte Wort `initialization`, gefolgt von einer Liste von Ausdrücken, welche die Unit initialisieren.

Initialisierungsteile von Units innerhalb eines Programms werden in der gleichen Reihenfolge ausgeführt, in der die Units in der `Uses`-Klausel des Hauptprogramms aufgeführt sind.

 Der **Finalization-Teil** ist optional und kann nur erscheinen, wenn eine Unit auch einen Initialisierungsteil enthält. Der Finalization-Teil besteht aus dem reservierten Wort `finalization`, gefolgt von einer Liste von Ausdrücken, welche die Unit beenden. `Finalization` ist das Gegenteil zur Initialisierung, und jede Ressource (Speicher, Dateien usw.), die von einer Unit während seiner Initialisierung erworben wird, werden typischerweise im Finalization-Teil wieder freigegeben.

 Sowohl Initialisierungsteil wie Finalization-Teil sind optional. Unser Beispielprogramm enthält weder das eine noch das andere.



### 3.4 Indirekte und zirkuläre Referenzen

Im allgemeinen sind Units über mehrere Stufen miteinander verknüpft. Im folgenden sehen wir uns zwei Spezialfälle an:

#### 3.4.1 Indirekte Unit-Referenzen


Die Uses-Klausel eines Moduls muss nur die Namen der Units enthalten, die dieses Modul direkt verwendet. Da eine Unit A in ihrer Uses-Klausel die Unit B aufführen kann, und Unit B wiederum eine weitere Unit C in ihrer Uses-Klausel aufnehmen kann, ist A indirekt auch von C abhängig. Ganze Kaskaden von indirekten Unit-Referenzen sind möglich. Um ein Modul zu kompilieren, muss der Compiler in der Lage sein, alle Units aufzufinden, von denen ein Modul direkt oder indirekt abhängt.

Wenn Sie am Interface-Teil einer Unit Veränderungen vornehmen, dann müssen Sie alle Units rekompilieren, die diese geänderte Unit verwenden. Wenn Sie PROJECT - BUILD ALL verwenden, so übernimmt der Compiler das für Sie.

Wenn Veränderungen nur am Implementations- oder Initialisierungsteil vorgenommen werden, so reicht es aus, die veränderte Unit erneut zu kompilieren.

Delphi erkennt, dass sich ein Interface-Teil einer Unit verändert hat, da während des Kompilierens eine Versionsnummer der Unit berechnet wird.

#### 3.4.2 Zirkuläre Unit-Referenzen

 **Zirkuläre Unit-Referenzen** treten auf, wenn Sie wechselseitig abhängige Units haben, d.h. wenn eine Unit A in ihrer Uses-Klausel Unit B aufführt, und wenn die Uses-Klausel von Unit B ihrerseits Unit A referenziert. Solche zirkulären Referenzen sind dann zugelassen, wenn höchstens eine der zwei Referenzen im Interface-Teil erscheint.

Wenn die Unit A in der Uses-Klausel des Interface-Teils der Unit B erscheint, und die Uses-Klausel im Interface-Teil der Unit B die Unit A aufführt, dann erzeugt Delphi eine Fehlermeldung wegen zirkulärer Unit-Referenzen.

Wechselseitig abhängige Units können in speziellen Situationen nützlich sein, aber verwenden Sie sie mit Bedacht. Wenn Sie unnötigerweise Gebrauch davon machen, dann machen sie Ihr Programm schwieriger in der Wartung und eher anfällig für Fehler.

### 3.5 Rekapitulation

#### 3.5.1 Zusammenfassung

In diesem Kapitel haben Sie die grundlegende Struktur von Programmen und Units kennengelernt. Im Detail ging es um folgende Themen:

- Sie haben in einem Beispielprogramm ein einfaches Formular mit einer Schaltfläche erzeugt und eine Ereignisbehandlungsroutine geschrieben, mit der sich das Formular schliessen lässt.
- Sie haben den Code dieses Beispielprogramms analysiert und dabei die Syntax von Programmen und Units kennengelernt. Sie kennen die fünf Teile einer Unit, nämlich Unit-Kopf, Interface-Teil, Implementationsteil, Initialisierungsteil und Finalization-Teil.
- Im letzten Teil sind zirkuläre und indirekte Unit-Referenzen eingeführt worden.

#### 3.5.2 Kapiteltest

2) Richtig oder falsch?


- a) Von den fünf Teilen einer Unit ist nur der Finalization-Teil optional.
- b) Bei `strMeinName` und `strmeinname` handelt es sich um zwei verschiedene Variablen.
- c) Die Programmdatei und alle Unit-Dateien gliedern sich in einen Interface- und einen Implementationsteil.

2) Was passiert, wenn Sie folgenden Code kompilieren:

```
Unit uEins
interface
uses uZwei;
implementation
end.
```


```
Unit uZwei
interface
uses uEins;
implementation
end.
```

## 4 Visuelle Programmierung


 In Kapitel 2 haben Sie bereits die wichtigsten Elemente der Entwicklungsumgebung kennengelernt. Nun werden Sie Ihre Erfahrung im Umgang damit weiter schulen:

- Sie rufen sich die Philosophie von Windows-Applikationen in Erinnerung.
- Sie lernen einige Tricks, wie man mit Formularen und anderen Komponenten für eine Applikation rasch eine ansprechende Benutzeroberfläche zusammensetzt.
- Sie gewinnen einen Überblick über die wichtigsten Register und Komponenten der Komponentenpalette.
- Sie erfahren den Unterschied zwischen modalen und nichtmodalen Fenstern.
- Und schliesslich starten Sie mit einem Projekt, dass Sie auch in den weiteren Lektionen als Übungsbeispiel begleiten wird. In diesem Projekt entwickeln Sie einen rudimentären Editor.

### 4.1 Die Philosophie von Windows-Applikationen

 Die Windows-Umgebung benutzt **Fenster** als grundlegende Bausteine der Benutzerschnittstelle. Eigentlich sind alle visuellen Elemente, beispielsweise auch Schaltflächen, in Windows Fenster.

Bei der visuellen Programmierung wird der Applikationsentwicklung ein neuer Schritt vorangestellt. Bevor man nämlich mit der eigentlichen Kodierung beginnt, wird im Drag-und-Drop-Verfahren eine visuelle Benutzerschnittstelle gestaltet.

 Die Grundbausteine von Delphi-Applikationen sind **Formulare**. Formulare sind spezielle Fenster, die andere visuelle und nichtvisuelle Elemente enthalten können. Wie jedes Windows-Fenster enthalten Formulare mindestens verstellbare Ränder und eine Titelleiste mit Systemmenü und Schaltflächen, um das Formular zu minimieren, zu vergrössern oder zu schliessen.

Wenn Sie ein Projekt beginnen, dann stellt Delphi Ihnen ein neues, leeres Formular zur Verfügung. Sie fügen zu diesem Formular Komponenten hinzu, welche Interaktionen mit den späteren Benutzerinnen und Benutzern ermöglichen.

Die visuelle Programmierung ist eine Antwort auf den Wechsel von linear ablaufenden zu ereignisgesteuerten Applikationen. Auf der Entwicklungsebene entspricht dem der Übergang von der prozeduralen zur ereignisorientierten Programmierung.

Die folgende Tabelle listet einige der wichtigsten Unterschiede auf:

	<b><i>Prozedurale Programmierung</i></b>	<b><i>Ereignisgesteuerte Programmierung</i></b>
<b>Ablauf</b>	linear	ereignisgesteuert
<b>Ablauf bestimmt durch</b>	Programm	Benutzer/in
<b>Benutzerschnittstelle</b>	textbasiert	grafikbasiert
<b>Kontrolle Environment</b>	Entwickler/in	Windows

## 4.2 Formulardesign

In diesem Abschnitt lernen Sie ein paar Tips und Tricks kennen, wie Sie die Gestaltung eines Formulars mit Hilfe von Komponenten rationalisieren können. Ausgiebigere Informationen finden Sie (wie immer) im Register INHALT der Online-Hilfe unter USING DELPHI - BASIC SKILLS - WORKING WITH COMPONENTS.

### 4.2.1 Eine Komponente auf einem Formular platzieren

Sie haben bereits gelernt, wie man eine Schaltfläche auf einem Formular platziert. Das Vorgehen für andere visuelle Komponenten ist ähnlich und umfasst immer die folgenden Schritte:

1. Wählen Sie in der Komponentenpalette das Register mit der gewünschten Komponente.
2. Selektieren Sie die Komponente, indem Sie darauf klicken.
3. Klicken Sie mit der Maus auf jene Stelle des Formulars, wo Sie die linke obere Ecke der Komponente haben möchten, und ziehen Sie mit gedrückter Maustaste nach unten rechts, bis die gewünschte Grösse erreicht ist.
4. Wenn Sie mit der Platzierung noch nicht zufrieden sind, können Sie die Komponente nachträglich mit gedrückter

Maustaste an jeden beliebigen Ort versetzen. Die Grösse lässt sich mit den Ziehmarken anpassen.

#### 4.2.2 Mehrere Komponenten platzieren

Wenn Sie mehrere gleichartige Komponenten auf einem Formular benötigen, dann müssen Sie die Anweisung oben nur für die erste Komponente befolgen. Alle weiteren können Sie mit EDIT - COPY (CTRL C) kopieren und mit EDIT - PASTE (CTRL V) einfügen.



1. Wählen Sie FILE - NEW. Sie erhalten das Dialogfenster NEW ITEMS. Versichern Sie sich, dass APPLICATION im Register NEW ausgewählt ist, bevor Sie OK drücken. Wie beim Start von Delphi erhalten Sie ein neues Projekt mit einem leeren Formular.
2. Wählen Sie im Register STANDARD die bereits bekannte Button-Komponente und platzieren Sie sie in der oben beschriebenen Art im Formular.
3. Drücken Sie CTRL C.
4. Drücken Sie anschliessend CTRL V.
5. Bewegen Sie die kopierte Schaltfläche an den gewünschten Ort im Formular.
6. Wiederholen Sie Schritt 4 und 5.

Sie haben jetzt ein Formular mit drei Schaltflächen. Delphi hat sie selbständig Button1, Button2 und Button3 benannt und führt sie unter diesen Namen in der Liste des Objektselektors. Beschriftet sind allerdings alle drei mit „Button1“.

Es gibt noch eine zweite elegante Art, mehrere gleichartige Komponenten in einem Formular zu platzieren: Wenn Sie bei der Auswahl der Komponente in der Komponentenpalette die UMSCHALT-Taste gedrückt halten, dann erhält das Symbol der Komponente einen blauen Rand und jedes Klicken ins Formular erzeugt von da an ein neues Exemplar dieser Komponente. Aufheben können Sie diese Mehrfachfunktion, wenn Sie in die Komponentenpalette klicken.

Sie können bestimmte Aktionen an mehreren Komponenten gleichzeitig vollziehen. Dazu müssen Sie aber erst mehrere Komponenten auswählen.



Klicken Sie auf Button1. Halten Sie die UMSCHALT-Taste gedrückt und klicken Sie auf Button2 und dann auf Button3. Die grauen Rechtecke in den vier Ecken einer Komponente zeigen Ihnen, dass sie ausgewählt ist. Fahren Sie mit der Maus über eine der drei Tasten, drücken Sie die linke Maustaste und gehen Sie mit gedrückter Maustaste an eine andere Stelle. Hellgraue

Konturen zeigen Ihnen an, dass Sie jetzt alle drei Buttons verschieben. Erst wenn Sie die Maustaste loslassen, werden die Komponenten am neuen Ort abgesetzt.

Wie Sie eine Komponente mit Hilfe der Ausrichtungspalette (VIEW - ALIGNMENT PALETTE) im Formular zentrieren können, haben Sie bereits in den vorangehenden Kapiteln gesehen. Natürlich können Sie damit auch mehrere Komponenten ausrichten. Wenn Sie mit Text besser zurechtkommen als mit Symbolen, dann erreichen Sie denselben Effekt mit dem Dialogfenster ALIGNMENT aus dem Menü EDIT - ALIGN.

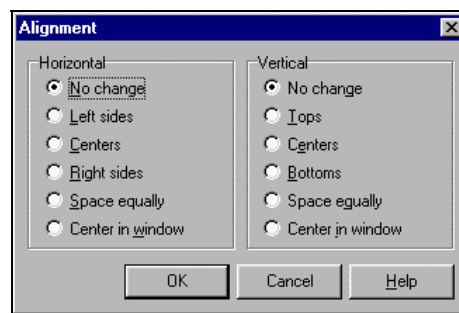


Abb. 14 Das Dialogfenster „ALIGNMENT“

Hier können Sie angeben, ob Sie mehrere Komponenten horizontal oder vertikal ausrichten möchten. Besonders praktisch ist die Option „SPACE EQUALLY“, mit der die Abstände zwischen den gewählten Komponenten ausgeglichen werden. Wenn Sie mehrere Komponenten selektiert haben, bezieht sich „CENTER IN WINDOW“ nicht auf jede einzelne Komponente, sondern auf die ganze Gruppe. In zwei Schritten können Sie also zuerst die Abstände zwischen mehreren Komponenten angleichen und dann die ganze Gruppe in der Mitte des Formulars plazieren.

Wenn Sie mehrere Komponenten selektiert haben, können Sie nicht nur alle gemeinsam verschieben, sondern Sie können auch Eigenschaften bei mehreren Komponenten gleichzeitig ändern. Wenn Sie verschiedene Arten von Komponenten gleichzeitig auswählen, zeigt der Objektinspektor nur noch jene Eigenschaften an, die allen gemeinsam sind.



Fügen Sie zu den drei bestehenden Buttons noch ein Label hinzu (3. Komponente von links im Register STANDARD). Wählen Sie mit der Umschalt-Taste alle vier Komponenten des Formulars aus. Klicken Sie im Objektinspektor auf das Pluszeichen links von der Eigenschaft FONT. Tragen Sie in der Wertespalte neben dem neu erschienenen SIZE den Wert 12 ein. Die Beschriftung aller ge-

wählten Komponenten sollte jetzt in der neuen Schriftgrösse erscheinen.

#### 4.2.3 Komponenten löschen

Eine oder mehrere Komponenten zu löschen, ist genau so einfach wie ihre Erzeugung. Sie selektieren eine oder mehrere Komponenten, die Sie löschen möchten und drücken die Taste DELETE. Wenn Sie dies rückgängig machen möchten, wählen Sie EDIT - UNDELETE oder CTRL Z.

Delphi entfernt nicht nur die Schaltflächen auf dem Formular, sondern auch die Einträge unter der Typendeklaration von TForm1 im Quellcode.



Selektieren Sie als erstes die eben erstellte Komponente Label und löschen Sie sie. Verschieben Sie dann Formular und Quelltexteditor so, dass der folgende Abschnitt des Quellcodes auch dann sichtbar ist, wenn das Formular aktiviert ist.



```
1 TForm1 = class(TForm)
2     Button3: TButton;
3     Button2: TButton;
4     Button1: TButton;
5     private...
```

Selektieren Sie die drei Schaltflächen und drücken Sie DELETE. Achten Sie darauf, wie die drei Einträge verschwinden und wieder erscheinen, wenn Sie jetzt CTRL Z drücken.

#### 4.2.4 Gruppieren von Komponenten

Ausser dem Formular selbst bietet Delphi diverse Komponenten - GroupBox, Panel, Notebook, TabbedNotebook und Scrollbox -, die andere Komponenten beinhalten können.



Diese werden häufig als **Container-Komponenten** bezeichnet. Sie können diese Container-Komponenten dazu verwenden, andere Komponenten zu gruppieren, so dass sie sich zur Entwurfszeit wie eine Einheit benehmen. So lassen sich z.B. Komponenten wie speed buttons (Mauspalettenschalter) und check boxes (Markierungsfelder) gruppieren, um dem Benutzer zusammengehörige Optionen zu bieten.

Bei der Plazierung von Komponenten innerhalb von Container-Komponenten erzeugen Sie eine neue Beziehung zwischen dem übergeordneten Container und den darin enthaltenen untergeordneten Komponenten. Operationen, die Sie zur Entwurfszeit an der (übergeordneten) Container-Komponente vornehmen, wie etwa Verschieben, Kopieren

oder Löschen, wirken sich auch auf jegliche darin gruppierte Komponenten aus.

Im allgemeinen fügen Sie zunächst die Container-Komponenten zum Formular hinzu, bevor Sie die zu gruppierenden Komponenten hinzufügen, da es am einfachsten ist, die zu gruppierenden Komponenten direkt aus der Komponentenpalette in die Container-Komponente einzufügen.

Bereits vorhandene Komponenten können Sie mittels Ausschneiden und Einfügen zu einer nachträglich erstellten Container-Komponente hinzufügen.

Sie werden dies nun ausprobieren, indem Sie Ihr Formular mit einer Symbolleiste und sechs speed buttons in der Art von Abb. 15 erweitern:

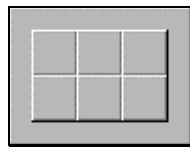


Abb. 15 Eine eigene Symbolleiste



1. Plazieren Sie die Komponente SPEEDBUTTON aus dem Register ADDITIONAL auf Ihrem Formular.
2. Fügen Sie dem Formular die Komponente PANEL aus dem Register STANDARD hinzu.
3. Selektieren Sie den ausserhalb des PANELS liegenden SPEEDBUTTON und schneiden Sie ihn mit EDIT - CUT oder CTRL X aus.
4. Selektieren Sie das PANEL und fügen Sie den ausgeschnittenen SPEEDBUTTON mit CTRL V darauf ein. Wiederholen Sie CTRL V fünf Mal, um fünf weitere SPEEDBUTTONS einzufügen.
5. Verschieben Sie die sechs SPEEDBUTTONS, bis sie einen kompakten, zweireihigen Block bilden.
6. Selektieren Sie alle sechs SPEEDBUTTONS gleichzeitig und platzieren Sie sie mit der Ausrichtungspalette in der Mitte des PANELS.

Bei dieser Übung ging es nur darum, gewisse Fertigkeiten im Umgang mit Komponenten zu üben. Sie werden sich deshalb in der Zen-Tugend des Gleichmuts üben und Ihr Projekt mit FILE - CLOSE ALL- SAVE CHANGES? - NO ins Nirwana schicken, statt es mit YES zu speichern. Sie wissen ja: Der Weg ist das Ziel!



#### 4.2.5 Owner und Parent

Bei der Gruppierung von Komponenten stösst man auf das Thema, welche Beziehungen zwischen verschiedenen Komponenten bestehen. Es gibt zwei wichtige Kategorien von Beziehungen:

- ☞ • Eignerschaft (**owner**): Wenn eine Komponente eine andere besitzt, so wird der Speicher dieser Komponente freigegeben, sobald der Speicher der besitzenden Komponente freigegeben wird. Ein Formular besitzt alle Komponenten, die sich auf diesem befinden. Das Formular wiederum gehört der Anwendung. Das bedeutet, dass alle Komponenten des Formulars zerstört werden, wenn ein Formular zerstört wird. Wenn schliesslich der Speicher der Anwendung selbst freigegeben wird, so wird auch der Speicher des Formulars (und all seiner Komponenten) freigegeben..
- ☞ • Über- und Unterordnung (**parent** und **child**): Die übergeordnete Komponente stellt grafisch eine Fläche zur Verfügung, innerhalb deren eine andere Komponente dargestellt wird. Eine untergeordnete Komponente kann nicht ausserhalb der Ränder der übergeordneten Komponente dargestellt werden.

Für Gruppierungen von Komponenten bedeutet dies, dass das Formular weiterhin der Besitzer aller Komponenten bleibt, und zwar unabhängig davon, ob diese innerhalb einer anderen gruppiert und damit jener Komponente untergeordnet sind. Für alle Komponenten, die nicht innerhalb einer anderen Komponente gruppiert sind, ist das Formular sowohl owner wie parent.

Eine weitere Art von Beziehung zwischen Komponenten werden Sie im Kapitel über objektorientierte Programmierung kennenlernen, nämlich jene zwischen Vorfahr oder

- ☞ englisch **ancestor** und Nachkomme oder **descendant**. Diese Begriffe beziehen sich auf die Vererbung zwischen Klassen, ihre Einführung erfolgt später und sie werden hier nur der Vollständigkeit halber erwähnt.

#### 4.3 Ereignisbehandlungsroutinen

Jede Komponente hat eine Reihe vorgegebener Ereignisse, auf die sie reagieren kann. Die Betonung liegt auf „kann“, denn eine Komponente reagiert nur auf Ereignisse, für welche Sie die entsprechenden Ereignisbehandlungsroutinen

geschrieben haben. Wie Sie bereits wissen, schreiben Sie eine Ereignisbehandlungsroutine, indem Sie die Komponente selektieren und im Objektinspektor das Feld neben dem gewünschten EVENT doppelklicken. Zwischen `begin` und `end` der automatisch erstellten Rumpfprozedur tippen Sie ihre eigenen Programmzeilen.

Auch dies probieren Sie wieder selbst aus. Sie werden ein Programm erstellen, das Ihnen die Position der Maus anzeigt, wenn Sie über das Formular fahren. In Delphi erreichen Sie dies, indem Sie nur eine einzige Zeile Code schreiben.



1. Öffnen Sie ein neues Projekt mit einem leeren Formular (falls Sie nicht mehr wissen, wie dies geht, hilft Seite 53 weiter).
2. Plazieren Sie eine Label-Komponente, die fast so breit ist wie das Formular, am oberen Rand. Nennen Sie sie *lblMausposition*.
3. Ändern Sie die Schriftgröße dieser Komponente auf 14.
4. Wählen Sie im Objektselektor das Formular Form1 aus und ändern Sie dessen Namen in *frmMauslauf*.
5. Wechseln Sie ins Register EVENTS und suchen Sie das Ereignis ONMOUSEMOVE. Doppelklicken Sie ins Feld daneben.
6. Geben Sie im Quelltexteditor an der Position des Cursors folgenden Code ein: `lblMausPosition.caption := 'Position: ' + InttoStr(X) + ' ' + InttoStr(Y);`
7. Testen Sie Ihr Programm mit RUN - RUN. Wenn Sie jetzt mit der Maus über das Formular fahren, zeigt das Label die entsprechende Mausposition an.
8. Speichern Sie das Programm in einem neuen Ordner «Prog4\_1», indem Sie die Unit «uMausPos.pas» und das Projekt «pApp4\_1.dpr» benennen.

Wenn Sie jetzt in den oberen Teil der Unit gehen, finden Sie dort eingerückt unter `Type TfrmMauslauf` die Zeile `procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,Y: Integer)`. Es handelt sich um die Deklaration der Ereignisbehandlungsroutine, mit der das Formular auf das Ereignis ONMOUSEMOVE reagiert. An den Einrückungen erkennt man, dass diese Prozedur zur Klasse `TfrmMauslauf` gehört. Komponenten sind nämlich in Delphi Klassen bzw. Objekte. Wie Sie bereits gehört haben, nennt man eine Prozedur oder Funktion, die zu einer Klasse gehört, **Methode**. Die Implementierung der Methode `FormMouseMove` befindet sich im Implementationsteil. Die Methodendeklaration im Interface-Teil und der Kopf der Me-

thode im Implementationsteil sind mit Ausnahme des Qualifizierers `TfrmMauslauf`. identisch.

Der Name des Komponententyps, gefolgt von einem Punkt, ist als **Qualifizierer** notwendig, um Methoden verschiedener Komponenten zu unterscheiden, die auf dieselben Ereignisse reagieren.

Funktionen, Prozeduren oder Methoden benötigen beim Aufruf häufig zusätzliche Informationen. Diese Informationen liefert man in Form von Parametern. In der Klammer nach dem Methodennamen folgt deshalb eine Liste der übergebenen Parameter.

An erster Stelle kommt der Parameter `Sender` vom Typ `TObject`, der Delphi darüber orientiert, welche Komponente das Ereignis empfangen und infolgedessen die Ereignisbehandlungsroutine aufgerufen hat.

Der zweite Parameter `Shift` gibt bei Tastatur- und Mausereignissen den Status der Maustasten und der Tasten ALT, CTRL und UMSCHALT an.

Die nächsten zwei Parameter `X` und `Y` geben die relative Position der Maus im Fenster an, wobei oben links 0, 0 ist. Diese zwei Parameter haben Sie verwendet, um die Mausposition anzuzeigen. Weil sie vom Typ `Integer` sind, lassen sie sich nicht direkt im `Caption` der Label-Komponente `lblMausPosition` anzeigen, sondern müssen zuerst mit der Funktion `IntToStr` in Strings umgewandelt werden.

Wie Sie gesehen haben, nimmt Delphi Ihnen bei der Erstellung von Behandlungsroutinen für Standardereignisse die Routinearbeit weitgehend ab, indem es den Prozedurkopf im Interface- und im Implementationsteil selbständig erstellt. Sie müssen nur den eigentlichen Code, mit dem das Programm auf bestimmte Ereignisse reagieren soll, eingeben.

#### 4.4 Die wichtigsten Komponenten

Bis jetzt haben wir nur mit wenigen visuellen Komponenten aus den Registern `STANDARD` und `ADDITIONAL` gearbeitet. Hier lernen Sie nun die meisten Komponenten aus der Komponentenpalette kennen.

Zu den restlichen Komponenten können Sie sich Hilfe holen, indem Sie entweder die gesuchte Komponente anklicken und dann F1 drücken oder (wenn Sie einen Überblick über alle Komponenten eines Registers möchten) in der Online-








Hilfe im Register INHALT „USING DELPHI - PROGRAMMING ENVIRONMENT - COMPONENT PALETTE- COMPONENT PALETTE PAGES“ öffnen und die gewünschte Seite anklicken. Sie sollten sich auf jeden Fall angewöhnen, mit F1 die Online-Hilfe anzuschauen, bevor Sie eine unbekannte Komponente verwenden. Sie erhalten dann jeweils einen Überblick über Eigenschaften, Methoden und Ereignisse sowie eine Anleitung (Tasks), wie Sie die Komponente praktisch verwenden.








#### 4.4.1 Die Komponenten des Registers Standard

Die Komponenten im Register STANDARD der Komponentpalette stellen Ihren Delphi-Anwendungen die Standardsteuerelemente der Windows-Oberfläche zur Verfügung.



Abb. 16 Das Register STANDARD

	MAINMENU	Menüleiste und ihre zugehörigen Dropdown-Menüs für ein Formular
	POPUWMENU	Popup-Menü, verfügbar für Formulare und Steuerelemente, wenn die Anwender/innen die Komponente mit der rechten Maustaste anklicken
	LABEL	Text, der sich nicht auswählen oder ändern lässt, z.B. ein Titel
	EDIT	Editierfeld, in dem der/die Benutzer/in eine einzelne Textzeile eingeben oder ändern kann
	MEMO	Editierfeld, in dem sich mehrere Textzeilen eingeben oder ändern lassen
	BUTTON	Aktionsschalter, den die Benutzer/innen betätigen, um bestimmte Aktionen auszulösen
	CHECKBOX	Markierungsfeld, mit dem eine Option selektiert oder die Markierung aufgehoben werden kann, um die Selektion der Option rückgängig zu machen; Checkboxes verwendet man, wenn mehrere Optionen, die sich gegenseitig nicht ausschliessen, in einer Gruppe zusammengefasst werden sollen




 <b>RADIOBUTTON</b>	Optionsschaltfeld, mit dem man eine Reihe sich gegenseitig ausschliessender Optionen im Gruppenfeld <code>TGroupBox</code> darstellen kann
 <b>LISTBOX</b>	Listenfeld, mit dessen Hilfe sich ein oder mehrere Listenelemente auswählen lassen
 <b>COMBOBOX</b>	Element, das ein Editierfeld mit einer Liste kombiniert; Anwender/innen können entweder Text im Editierfeld eingeben oder einen Eintrag aus der Liste selektieren
 <b>SCROLLBAR</b>	Windows-Bildlaufleiste zum Blättern in einem Fenster, Formular oder Steuerelement
 <b>GROUPBOX</b>	Standard-Windows-Gruppenfeld, mit der verwandte Elemente, z.B. Markierungsfelder, in einem Formular angeordnet werden können
 <b>RADIOGROUP</b>	Gruppenfeld, das die Aufgabe, Optionsschaltfelder zu gruppieren, vereinfacht
 <b>PANEL</b>	Bedienfeld, auf dem andere Steuerelemente, z.B. speed buttons eingefügt werden können







#### 4.4.2 Die Komponenten des Registers Additional

Die Komponenten im Register `ADDITIONAL` der Komponentenpalette realisieren eine Reihe spezialisierter Windows-Steuerelemente für Ihre Delphi Anwendungen.



Abb. 17 Das Register `ADDITIONAL`

 <b>BITBTN</b>	Schaltfläche, die zur Anzeige einer Bitmap dienen kann
 <b>SPEEDBUTTON</b>	Schaltfläche, die als Träger einer Bitmap dienen kann (dafür aber keine Beschriftung mit Text bietet); Speedbuttons werden (als mit Bitmaps dekorierte symbolische Abkürzungen für irgendwelche Befehle) zusammengefasst und fungieren dann als Komponenten einer Werkzeugleiste
 <b>MASKEDIT</b>	Editierfeld, das im Gegensatz zu <code>EDIT</code> die Möglichkeit bietet, besondere Formatangaben in bezug auf Dateneingabe und -anzeige festzulegen

 STRINGGRID	-anzeige festzulegen Gitterform, die Sie benutzen können, um Strings in Spalten und Zeilen anzuzeigen
 DRAWGRID	Gitterform, die Sie benutzen können, um Daten in Spalten und Zeilen anzuzeigen
 IMAGE	Anzeige von Bitmap-, Icon- oder Meta-Dateien
 SHAPE	Komponente, die geometrische Figuren zeichnet; zur Auswahl stehen Ellipse, Kreis, Rechteck und Quadrat mit und ohne abgerundete Ecken
 BEVEL	Linien oder Rahmen in einer dreidimensionalen, „modellierten“ Darstellung
 SCROLLBOX	in der Grösse veränderlicher Ausschnitt, der bei Bedarf automatisch Bildlaufleisten anzeigt

#### 4.4.3 Die Komponenten des Registers Win95

Im Register WIN95 der Komponentenpalette finden Sie für Ihre Delphi-Anwendungen die Standardsteuerelemente der Win95-Benutzerschnittstelle.

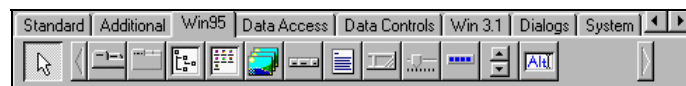














Abb. 18 Das Register WIN95

 TABCONTROL	einfache Registermenge, die sich mit einem Verteiler in einem Kartekasten oder Notizbuch vergleichen lässt; für ein mehrseitiges Dialogfeld eignet sich PAGECONTROL besser als TABCONTROL
 PAGECONTROL	eine Reihe von Seiten, die für die Erstellung mehrseitiger Dialogfelder verwendet wird
 TREEVIEW	Steuerung und Anzeige eines Strukturbaums, d.h. der logischen, hierarchischen Beziehung einer Reihe von Objekten
 LISTVIEW	Anzeige einer Liste in Spalten; kann Daten in verschiedensten Ansichten anzeigen
 IMAGELIST	Bilderliste, d.h. Sammlung von Bildern gleicher Grösse, von denen jedes über seinen Index erreicht werden kann

	HEADERCONTROL	Überschriftensteuerung, mit der Sie oberhalb von Text- oder Zahlenspalten eine Überschrift anzeigen können
	RICHEDIT	Steuerung für formatierte Memos im Rich-Text-Format
	STATUSBAR	Statusleiste zur Angabe des Zustands von Aktionen im unteren Teil des Bildschirms
	TRACKBAR	Steuerbalken mit Schieberegler zur Anzeige und Veränderung eines Wertes
	PROGRESSBAR	Fortschrittsanzeiger, mit dem man die Benutzer/innen über den Fortgang eines Ablaufs orientiert
	UPDOWN	Pfeiltasten für das Erhöhen und Vermindern von Werten
	HOTKEY	Steuerung, die ein Tastenkürzel mit einer Komponente verbindet

#### 4.4.4 Die Komponenten des Registers Data Access

Die Komponenten im Register DATA ACCESS der Komponentenpalette erlauben Ihren Delphi-Anwendungen den durch spezialisierte Steuerelemente kontrollierten Zugriff auf Datenbanken.

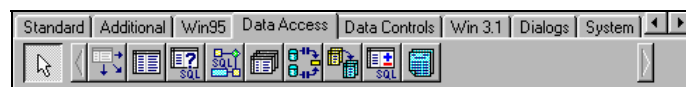


Abb. 19 Das Register DATA ACCESS

Weil Datenbankprogrammierung kein Thema dieses Kurses ist, sondern erst im Delphi-Aufbaukurs vorkommt, werden die Komponenten dieses Registers nicht einzeln vorgestellt.

#### 4.4.5 Die Komponenten des Registers Data Controls

Die Komponenten im Register DATA CONTROL der Komponentenpalette stellen Ihren Delphi-Anwendungen spezielle Datenbank-Steuerelemente bereit.



Abb. 20 Das Register DATA CONTROL

Für dieses Register gilt wie für das vorangehende, dass die einzelnen Komponenten nicht vorgestellt werden, weil sie nur für die nicht behandelte Datenbankprogrammierung notwendig sind.

#### 4.4.6 Die Komponenten des Registers Win 3.1

Die Komponenten im Register WIN 3.1 der Komponentenpalette stellt Ihren Delphi-Anwendungen Windows3.1-Steuer-elemente zur Verfügung.

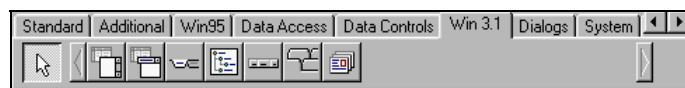


Abb. 21 Das Register WIN 3.1

Die Komponenten dieses Registers sichern vor allem die Kompatibilität mit Windows 3.1 und Delphi 1.0. Für Windows95- und NT-Anwendungen sollte man dieses Register nicht verwenden, sondern die neuen Komponenten im Register WIN95 oder DATACONTROLS benutzen. Die Win-3.1-Komponenten werden deshalb nicht einzeln vorgestellt. Falls Sie Informationen benötigen, können Sie wie bei jeder Komponente das Icon anklicken und F1 drücken, um die entsprechende Seite in der Online-Hilfe aufzurufen.

#### 4.4.7 Die Komponenten des Registers Dialogs

Die Komponenten im Register DIALOGS der Komponentenpalette stellen Ihren Delphi-Anwendungen die Standard-dialoge der Windows-Oberfläche zur Verfügung. Diese Standarddialoge bieten zusammen eine einheitliche Schnittstelle für die Dateiverwaltung, also für Operationen wie das Öffnen, Speichern und Drucken von Dateien.











Abb. 22 Das Register DIALOGS



Ein Standarddialogfenster wird geöffnet, wenn dessen Methode `Execute` aufgerufen wird.

Jede der Standarddialogkomponenten (ausser der Komponente zur Druckereinrichtung - `PRINTERSETUP`) hat eine `OPTIONS` genannte Gruppe von Eigenschaften, die Sie mit dem Objektinspektor bearbeiten können. Die Eigenschaften der Options-Gruppe betreffen Erscheinungsbild und Verhalten der Standarddialoge. Sie können die einzelnen Eigenschaften und ihre Belegung durch einen Doppelklick auf `OPTIONS` im Fenster des Objektinspektors anzeigen lassen.

	OPENDIALOG	Windows-Standarddialog zum Öffnen einer Datei
	SAVEDIALOG	Windows-Standarddialog zum Speichern einer Datei
	FONTDIALOG	Windows-Standarddialog zur Auswahl von Schriftarten
	COLORDIALOG	Windows-Standarddialog zur Auswahl von Farben
	PRINTDIALOG	Windows-Standarddialog zur Steuerung des Druckvorgangs
	PRINTERSETUPDIALOG	Windows-Standarddialog zur Druckereinrichtung
	FINDDIALOG	Windows-Standarddialog zur Suche von Text in einer Datei
	REPLACEDIALOG	Windows-Standarddialog zum Ersetzen eines gesuchten Textstrings

#### 4.4.8 Die Komponenten des Registers System

Die Komponenten im Register `SYSTEM` der Komponentpalette stellen Ihren Delphi-Anwendungen bestimmte Steuerelemente zur Systemkontrolle zur Verfügung.

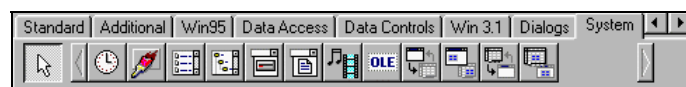









Abb. 23 Das Register SYSTEM

Im Register `System` lernen Sie nur die erste Hälfte der Komponenten kennen, denn die Komponenten auf der rech-

ten Seite dienen für DDE und OLE. Diese zwei Themen behandelt erst der Aufbaukurs.

 TIMER	nicht sichtbare Komponente, welche die Verknüpfung von Ereignissen mit zeitlichen Intervallen erlaubt
 PAINTBOX	rechteckiger Bereich innerhalb eines Formulars zur Abgrenzung für (in diesem Bereich) geplante Zeichen- bzw. Malaktionen durch die Anwendung
 FILELISTBOX	Anzeige einer Liste der Dateien im aktuellen Verzeichnis
 DIRECTORYLISTBOX	Anzeige der Verzeichnisstruktur des aktuellen Laufwerks, damit die Benutzer/innen das aktuelle Verzeichnis wechseln können
 DRIVECOMBOBOX	Anzeige einer Liste der aktuellen definierten Laufwerke
 FILTERCOMBOBOX	Filter bzw. Auswahlmaske, um nur ausgewählte Dateien in der Liste anzuzeigen
 MEDIAPLAYER	Steuerungseinheit im VCR-Stil, mit deren Hilfe sich Video- und Audio-Dateien abspielen lassen

#### 4.4.9 Weitere Register und Komponenten

Alle weiteren Register beinhalten Komponenten, die erst im fortgeschrittenen Stadium interessant sind, oder Beispiele. Sie werden nicht näher vorgestellt.

### 4.5 Das Editor-Projekt

„Lang ist der Weg durch Lehren, kurz und wirksam durch Beispiele“ (Seneca, römischer Dichter und Philosoph, 4 v.Chr. - 65 n.Chr.)

Als nächste Übung werden Sie einen kleinen Texteditor bauen. Sie schliessen dabei nähere Bekanntschaft mit einigen der vorgestellten Komponenten. Im Verlauf des Kurses werden Sie dieses Projekt mit weiteren Routinen, Tests und Experimenten ausbauen.



1. Starten Sie ein neues Projekt mit einem Standardformular (FILE - NEW APPLICATION).
2. Klicken Sie im Register ADDITIONAL der Komponentenpalette mit der Maus auf BITBTN und anschliessend ins Formular.

3. Verschieben Sie den Button nach unten rechts im Form. Jede Komponente lässt sich auch präzise mit der Tastatur verschieben, indem Sie die CTRL-Taste gedrückt halten und mit den Cursorpfeilen die Richtung angeben.
4. Im Objektinspektor links klicken Sie auf Name und überschreiben Button mit *btnbitClose*.
5. Anschliessend suchen Sie im PROPERTY-Register die Eigenschaft KIND und holen aus der Liste *bkClose*. Damit haben Sie einen vordefinierten Bitmap-Button geholt, der dazu dient, ein Formular zu schliessen.
6. Unter Caption ersetzen Sie *&Close* mit *&Ende*. Das Zeichen & definiert ein Tastenkürzel, so dass der Editor sich später auch mit ALT E schliessen lässt.
7. Nun ist es Zeit, das Projekt mit FILE - SAVE ALL zu speichern. Erstellen Sie im Verzeichnis mit den Beispielprogrammen einen neuen Ordner «Editor» und nennen Sie die Unit «uMainEdi.pas» und das Projekt «pEditor.dpr».
8. Fügen Sie als nächstes die Komponente MEMO aus der Palette STANDARD hinzu.
9. Ändern Sie die Eigenschaft ALIGN auf *alTop* und vergrössern Sie das Memofeld nach unten bis zum Button. Ändern Sie die Eigenschaft SCROLLBARS auf *ssVertical*.
10. Klicken Sie auf die Eigenschaft LINES und danach auf den kleinen punktierten Knopf auf derselben Zeile. Im Property-Editor, der sich darauf öffnet, löschen Sie den Text „memEditor“ und schliessen dann mit OK ab.
11. Fügen Sie aus dem Register STANDARD zwei Buttons in den unteren Teil des Formulars ein, benennen Sie sie mit *btnOpen* und *btnSave* und beschriften Sie die Schaltflächen mit *&Laden* und *&Speichern*.
12. Fügen Sie aus dem Register DIALOGS die beiden Komponenten OPENDIALOG und SAVEDIALOG hinzu. Nennen Sie sie *dlgOpenText* und *dlgSaveText*. Diese Komponenten nennt man nicht-visuell, da sie als Schnittstelle dienen und somit zur Laufzeit nicht sichtbar sind.
13. Markieren Sie beide Dialogkomponenten mit Hilfe der Umschalt-Taste. Ändern Sie dann für beide gleichzeitig die Eigenschaft FILTER, indem Sie *Textdateien|.txt* in die Wertespalte eintragen.
14. Jetzt wollen wir herausfinden, welche Prozedur hinter den Knopf Laden zu stehen kommt. Klicken Sie auf die Open-Dialog-Komponente und drücken Sie F1. Klicken Sie dort auf EXECUTE und darin wiederum auf EXAMPLE. Hier erhalten Sie ein Beispiel, wie man eine Textdatei öffnet. Markieren Sie die

ganze Bedingung `if Open...end;` und drücken Sie CTRL C. Sie haben nun das Beispiel in der Zwischenablage.

15. Verlassen Sie die Hilfe. Doppelklicken Sie auf den Laden-Button, worauf Sie im Quelltexteditor eine Rumpfprozedur erhalten. Fügen Sie den Beispielcode mit CTRL V aus dem Clipboard ein. Da wir die Namen unserer Komponenten geändert haben, müssen wir beim Code einige Anpassungen machen: `OpenDialog1` ersetzen wir mit `SEARCH - REPLACE` durch `dlgOpenText`. `Memo1` wechseln wir mit `memEditor` und `SaveDialog1` mit `dlgSaveText`.

16. Doppelklicken Sie im Formular auf die Schaltfläche Speichern und schreiben Sie folgende Zeile:

```
if dlgSaveText.Execute then
```

```
    memEditor.Lines.SaveToFile(dlgSaveText.FileName);
```

17. In der `SaveDialog`-Komponente können Sie die Eigenschaft `OPTIONS` durch Doppelklicken vergrößern, um die einzelnen Felder ein- und auszuschalten. Stellen Sie so sicher, dass vor dem Überschreiben einer Textdatei eine Kontrollfrage erscheint.

18. Mit F9 testen Sie Ihr Programm. Ihr Windows-Programm läuft nun ab, bis Sie es mit ENDE oder ALT E schliessen.

Es gibt nun verschiedene Techniken, dieses Programm weiter auszubauen und zu verbessern, wie etwa die Verwendung eines Menüs, von Suchroutinen, Ausnahmebehandlungen, oder Zugriffsroutinen auf die wir im Verlauf dieses Kurses im Zusammenhang mit Object Pascal noch zu sprechen kommen.

#### 4.6 Modale und nichtmodale Dialogfenster

Am Editor-Projekt lässt sich auch gerade der Unterschied zwischen modalen und nichtmodalen Dialogfenstern erklären. Die zwei Komponenten `OPENDIALOG` und `SAVEDIALOG` öffnen nämlich zwei modale Dialogfenster. Ein



**modales Dialogfenster** muss geschlossen werden, bevor man wieder zurück zum aufrufenden Fenster gelangt. Sie können dies ausprobieren, indem Sie Ihr Editor-Programm laufen lassen, den Speichern-Button betätigen und versuchen, das Hauptformular anzuklicken, solange der Speicherdialog geöffnet ist. Wie Sie sehen, ist ein Wechsel zwischen den Fenstern, den Sie aus anderen Anwendungen kennen, mit einem modalen Fenster nicht möglich. Dies gilt allerdings nur für die Fenster des Editor-Programms, in ein

Fenster einer anderen geöffneten Anwendung können Sie jederzeit wechseln.



Ein **nichtmodales Dialogfenster** oder Formular kann dagegen auf dem Bildschirm bleiben, während die Benutzer/innen in einem anderen Formular arbeiten. Nichtmodale Formulare ruft man mit der Methode `Show` auf, modale Fenster dagegen mit `ShowModal`.

## 4.7 Rekapitulation

### 4.7.1 Kapiteltest

- 1) Ist die folgende Aussage richtig oder falsch: Mehrere markierte Komponenten lassen sich gemeinsam bewegen, löschen oder in ihrer Grösse verändern.
- 2) In Abb. 24 sehen Sie ein Formular A mit einem Panel B, zwei speed buttons C und D sowie einem Button E.

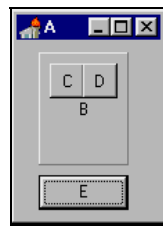


Abb. 24 Owner und parents

Suchen Sie aus den folgenden Aussagen alle richtigen heraus.

- a) Das Panel B ist owner und parent der speed buttons C und D.
  - b) Das Formular A ist owner von B, C, D und E und parent von B und E.
  - c) Der Button E ist ein child von A.
  - d) Das Panel B ist parent von C und D.
- 3) Welches Register in der Komponentenpalette enthält die Komponente `TABBEDNOTEBOOK`?


#### 4.7.2 Übung

##### Übung 4\_2

Erstellen Sie auf einem neuen Formular vier gleich grosse Buttons. Benutzen Sie die Ausrichtungspalette oder das Dialogfenster ALIGNMENT, um die vier Button in einer Reihe auszurichten und die Abstände gleich gross zu machen. Schieben Sie die ganze Gruppe an den unteren Formularrand und zentrieren Sie sie horizontal in der Mitte des Formulars.


## 5 Konstanten und Variablen

Während Sie bis jetzt vor allem Delphi als Entwicklungsumgebung kennengelernt haben, folgt in den nächsten Kapiteln eine Einführung in die Sprache Object Pascal. Wenn Sie mit strukturierter Programmierung vertraut sind, oder Turbo Pascal kennen, wird dies grösstenteils eine Repetition für Sie sein.

 Dieses Kapitel fängt mit den grundlegenden Elementen der Programmiersprache an, nämlich mit Konstanten und Variablen. Sie lernen,

- dass ein Programm sich aus Daten und Routinen zusammensetzt,
- was Konstanten sind, und wie Sie sie deklarieren,
- was Variablen sind, und wie Sie sie deklarieren,
- wie Sie Variablen mit Hilfe des Zuweisungsoperators Werte zuweisen
- und was typisierte Konstanten sind.

### 5.1 Daten und Routinen

 Jede Programmiersprache setzt sich aus einer begrenzten Menge von Sprachelementen zusammen. Diese Sprachelemente lassen sich grundsätzlich einmal einerseits in **Daten** und andererseits in **Routinen** zur Verarbeitung der Daten einteilen. Zu den Daten gehören Konstanten und Variablen, zu den Routinen dagegen Funktionen, Prozeduren und Methoden.

Wenn wir das Beispiel einer Addition nehmen, dann sind die Zahlen Daten und die Methodik, um sie zusammenzuzählen, die Routine.

Objekte oder Klassen, die wir später behandeln werden, sind Zwitter, denn sie verbinden Daten und die zu den Daten gehörigen Routinen.

In diesem und dem nächsten Kapitel wenden wir uns als erstes den Daten und ihren Typen zu. Verschiedene Arten von Routinen lernen Sie dann in den folgenden Kapiteln kennen. Und das Kapitel Klassen und Instanzen erhält schliesslich, was Klassen, Objekte und Instanzen sind.

## 5.2 Grundelemente von Object Pascal

Bevor Sie Konstanten und Variablen kennenlernen, ist eine kurze Einführung in einige Grundeinheiten der Sprachsyntax notwendig, denn alle weiteren Sprachelemente bauen sich aus diesen Grundelementen auf.

### 5.2.1 Tokens

☞ **Tokens** sind die kleinste sinnvolle Texteinheit in Object Pascal. Tokens sind:

- Zeichenketten (Strings)
- Bezeichner
- Labels
- Zahlen
- Reservierte Wörter
- Spezielle Symbole, z.B. Operatoren

Nur zur Erinnerung: Bezeichner und ihre Namenskonventionen sind schon in Kapitel 2.3.2 eingeführt worden.

### 5.2.2 Operatoren

☞ **Operatoren** sind Symbole, z.B. +, - und <, oder reservierte Wörter wie `div` und `or`, die anzeigen, dass einige Operationen auf ein oder mehrere Daten angewandt werden. Da sich Operatoren in Ausdrücken kombinieren lassen, gibt es eine Rangfolge, die festlegt, welche Operationen zuerst ausgeführt werden. Diese Rangfolge finden Sie im Index der Online-Hilfe unter dem Stichwort „precedence of operators“.

☞ In Object Pascal sind die meisten Operatoren **binär**; sie verknüpfen zwei Operanden. Die restlichen Operatoren arbeiten mit nur einem Operanden und werden daher als **unär** bezeichnet. Binäre Operatoren werden mit der normalen algebraischen Schreibweise, z.B.  $A+B$ , benutzt. Ein unärer Operator steht immer unmittelbar vor seinem Operanden, wie z.B. bei  $-B$ .

### 5.2.3 Ausdrücke

☞ **Ausdrücke** (englisch expressions) sind eine Kombination von Operatoren und Operanden, die zu einem einzigen Wert führen. Diese Operanden können Konstanten, Funktions-



aufrufe, Prozedur-Anweisungen, Mengen-Konstruktoren oder Variablen sein.

## 5.3 Konstanten

### 5.3.1 Begriffsdefinition

☞ Eine **Konstante** verwenden Sie, wenn Sie in Ihrem Programm immer wieder auf Strings, Zahlen oder andere Werte zugreifen wollen, die sich nicht ändern. Statt diese Werte jedes Mal im Code neu einzugeben, deklarieren Sie eine Konstante, d.h. einen Bezeichner, der den Wert repräsentiert.

Konstanten zeichnen sich, wie der Name es schon sagt, dadurch aus, dass sie während der Ausführung eines Programms konstant bleiben.

Es gibt vor allem zwei Gründe, Konstanten zu verwenden:

1. Der Programmcode wird verständlicher, wenn man statt nackten Zahlen sprechende Bezeichner verwendet, also z.B. `AnzahlFelder` statt `64`.
2. Oft ist es sinnvoll, nicht nur universelle Werte wie `Pi` durch Konstanten auszutauschen, sondern auch häufig verwendete Werte eines Programms durch Konstanten zu ersetzen. Das Programm wird dadurch besser wartbar, denn falls ein häufig verwendeter Wert sich ändert, müssen Sie den neuen Wert nur in der Konstantendeklaration ersetzen, statt an jedem Ort, wo dieser Wert verwendet wird.

### 5.3.2 Konstantendeklaration

Konstanten deklariert man im Deklarationsteil nach dem reservierten Wort `const`. In Kapitel 3 haben Sie gehört, dass Deklarationen normalerweise am Anfang von Interface- und Implementationsteil einer Unit stehen. Wie Sie später noch sehen werden, kann es auch innerhalb von Prozeduren und Funktionen wieder einen Deklarationsteil geben.

Die generelle Syntax einer Konstantendeklaration sieht folgendermassen aus:

```

const
  Konstantenname1 = Konstantenwert1;
  Konstantenname2 = Konstantenwert2;
  Konstantenname3 =
    Konstantewert1 * Konstantenwert2;

```

Das reservierte Wort `const` leitet den Deklarationsteil ein. Dann folgen die einzelnen Konstantendeklarationen mit Bezeichner, Gleichzeichen und Wert. Ein Strichpunkt schliesst jede einzelne Deklaration ab. Bei der Vergabe von Bezeichnern unterliegen Sie, wie Sie bereits auf Seite 27 gesehen haben, gewissen Einschränkungen. Beachten Sie, dass Konstantendeklarationen im Gegensatz zu Variablenzuweisungen das Gleichzeichen verwenden. Ausserdem können Sie Konstanten nicht nur direkt einen Wert zuweisen, sondern auch das Resultat aus Standardoperationen mit bereits definierten Konstanten. Eine konkrete Konstantendeklaration könnte so aussehen:

```

const
  Pi           = 3.1415926536;
  MaxBreite    = 640;
  MaxHoehe     = 480;
  Vorname      = 'Silvia';
  Nachname     = 'Rothen';
  GanzerName   = Vorname + ' ' + Nachname;

```

Die zweite und dritte Konstante werden Sie jetzt benutzen, um die maximale Grösse des Editors aus dem vorangehenden Kapitel auf das VGA-Format von 640 Pixeln Breite und 480 Pixeln Höhe zu beschränken. Da diese Werte nur in der Unit «uMainEdi» gültig sein sollen, fügen wir den Deklarationsteil unterhalb von `implementation` ein.



Ergänzen Sie die Unit «uMainEdi» nach den Zeilen `implementation` und `{ $R *.DFM }` mit folgendem Code:

```

const
  MaxBreite    = 640;
  MaxHoehe     = 480;

```

Wählen Sie anschliessend im Objektselektor das Formular *frmEditor* aus. Wechseln Sie ins Register `EVENTS` und doppelklicken Sie neben `ONRESIZE`. Fügen Sie nun im Quelltexteditor zwischen `begin` und `end` die folgenden zwei Anweisungen ein:

```

if frmEditor.Height > MaxHoehe
  then frmEditor.Height := MaxHoehe;

if frmEditor.Width > MaxBreite
  then frmEditor.Width := MaxBreite;

```

Wenn Sie nun Ihr Programm mit F9 ausführen und mit der Maus die Ränder des Formulars gegen aussen ziehen, dann lässt es sich maximal auf das Format 640 x 480 vergrössern.

Konstanten haben die folgenden wichtigen Eigenschaften:

1. Der Wert einer Konstanten lässt sich zur Laufzeit nicht ändern.
2. Die Deklaration von Konstanten hat keinen negativen Einfluss auf die Performance eines Programms, denn der Compiler setzt bei der Erzeugung der ausführbaren Exe-Datei für die Konstanten die entsprechenden Werte ein.
3. Aus der letzten Bemerkung folgt, dass Konstanten keine Speicheradresse haben. Es handelt sich nur um eine praktische Art, während der Entwicklung mehrmals verwendete Werte und Daten durch einen Namen zu ersetzen.

## 5.4 Variablen

### 5.4.1 Begriffsdefinition



Eine **Variable** ist ein Bezeichner für einen veränderbaren Wert. Technisch stellt eine Variable eine Speicheradresse dar, deren Inhalt sich während der Laufzeit ändern kann. Object Pascal ist eine streng typisierte Sprache. Eine Variable kann deshalb nicht beliebige Werte enthalten, sondern nur solche eines bestimmten Typs.

Eine Variable hat vier Charakteristiken:

1. **Bezeichner** (Variablenname) : Der Bezeichner identifiziert die Variable, so dass Sie im Programm nicht mit Speicheradressen hantieren müssen, sondern mit Hilfe eines möglichst sprechenden Variablennamens auf Werte zugreifen können.
2. **Typ**: Der Typ bestimmt, welche Werte für die Variable zulässig sind und welche Operationen mit diesen Werten ausgeführt werden können.
3. **Wert**: Der Wert ist die Information, die an der Speicheradresse der Variablen gespeichert ist. Auf Maschinenebene ist dies nur eine Abfolge von Null und Eins. Erst durch den Typ lässt sich eine solche Abfolge als Buchstabe A oder als Zahl 65 interpretieren.

**4. Speicherplatz und Speicheradresse:** Für jede Variable wird im Arbeitsspeicher unter einer eindeutigen Adresse Platz reserviert. Der Typ bestimmt die Grösse des reservierten Speicherplatzes. Dank Variablen nimmt der Compiler Ihnen die mühsame Arbeit ab, direkt mit Speicheradressen zu hantieren.

#### 5.4.2 Variablendeklaration

Auch Variablen deklariert man in einem Deklarationsteil. Sie werden mit dem reservierten Wort `var` eingeleitet. Die generelle Syntax einer Variablendeklaration sieht folgendermassen aus:

```
var
  Variablenname1           : Type1;
  Variablenname2, Variablenname2 : Type2;
  Variablenname3           : Type3;
```

Das reservierte Wort `var` leitet einen Variablendeklarationsteil ein. Die einzelnen Variablendeklarationen setzen sich aus Bezeichner, Doppelpunkt und Typ zusammen. Auch diese Deklarationen schliesst der in Pascal obligate Strichpunkt ab. Wenn Sie in einem Deklarationsblock mehrere Variablen des gleichen Typs definieren möchten, dann ist es nicht nötig, für jede eine einzelne Deklaration zu schreiben. Sie können stattdessen wie in der mittleren Zeile die Variablen durch Kommas trennen und ihnen dann gemeinsam einen Typ zuweisen.

Eine konkrete Variablendeklaration könnte so aussehen:

```
var
  Formhoehe, Formbreite : integer;
  Kundenname : string[20];
```

☞ Eine Variablendeklaration reserviert erst Speicherplatz, weist aber im Gegensatz zu einer Konstantendeklaration noch keinen Wert zu. Nach der Deklaration ist somit eine Variable immer noch undefiniert. Bevor man im Code auf sie zugreifen kann, muss man sie also **initialisieren**, d.h. einen Wert zuweisen. Der Compiler fängt eine solch fehlende Initialisierung nicht ab, was zu unangenehmen Fehlern im Programm führen kann.

#### 5.4.3 Variablenzuweisungen

Eine **Zuweisung** weist dem Bezeichner auf der linken Seite des **Zuweisungsoperators** „:=“ den Wert des Ausdrucks auf der rechten Seite zu. Zugewiesen werden Werte, Kon-

stanten, andere Variablen oder Resultate aus Operationen, sofern sie mit dem Typ der Variablen zuweisungskompatibel sind.

Dies ist die allgemeine Form einer Variablenzuweisung:

```
Variablenname1 := 5;
Variablenname2 := Konstantenname1;
Variablenname3 := Variablenname2;
Variablenname4 := 2 * Variablenname3;
```

Variablendeklaration und Zuweisung werden Sie nun anhand des Editor-Projekts ausprobieren. Wir ergänzen den Editor so, dass im Textfenster zu Beginn der String „Willkommen im selbstgebauten Editor“ erscheint.



Gehen Sie ins Editor-Projekt. Wählen Sie im Objektselektor das Formular `memEditor`. Doppelklicken Sie im Register EVENTS in die Wertespalte neben `OnCreate`. Die Ereignisbehandlungsroutine, die Sie nun schreiben werden, wird immer als erstes ausgeführt, sobald das Programm gestartet und das Hauptformular erzeugt wird. Die Routine zur Behandlung des Ereignisses `ONCREATE` des Hauptformulars eignet sich für allgemeine Initialisierungen, z.B. von globalen Variablen, weil sie zuerst ausgeführt wird.

Ergänzen Sie die Rumpfprozedur, so dass Sie den folgenden Code erhalten:

```
1 procedure TfrmEditor.FormCreate
2   (Sender: TObject);
3   var
4     strWillkommen      : string[50];
5   begin
6     memEditor.Lines.Add(strWillkommen);
7   end;
```

Führen Sie diesen Code mit F9 aus.

Bei der Ausführung erscheint nun im Textfeld eventuell nichts, eventuell ein beliebiger String. Wir haben nämlich genau das gemacht, vor dem das letzte Unterkapitel gewarnt hat. In Zeile 3 und 4 haben wir eine Variable deklariert und in Zeile 6 haben wir sie den Zeilen der Memokomponente zugewiesen, ohne dass sie initialisiert wurde. Wie Sie sehen, führt dies weder zu einer Fehlermeldung beim Kompilieren noch zu einem Laufzeitfehler.




Mit den folgenden kursiv markierten Ergänzungen korrigieren Sie Ihren Fehler:

```
1 procedure TfrmEditor.FormCreate
2   (Sender: TObject);
3   const
4     Willkommenstext =
```

```

5      'Willkommen im selbstgebauten Editor';
6  var
7      strWillkommen      : string[50];
8  begin
9      strWillkommen := Willkommenstext;
10     memEditor.Lines.Add(strWillkommen);
11 end;

```


 Bevor Sie den Inhalt der Variable den Zeilen der Memo-komponente anfügen, initialisieren Sie sie in Zeile 9. Und zwar weisen Sie der Variable nicht direkt einen Wert zu, sondern den Wert der Konstanten `Willkommenstext`, die Sie in den Zeilen 3 bis 5 deklariert haben. Natürlich könnten wir wie in der folgenden Zeile auch direkt einen String statt Konstanten und Variablen verwenden.

```


memEditor.Lines.Add
    ('Willkommen im selbstgebauten Editor');

```

Aber ich wollte Ihnen ja ein Beispiel für Variablendeklarationen und Zuweisungen liefern.

 Bei Zuweisungen muss man unbedingt beachten, dass der Wert **zuweisungskompatibel** mit dem Typ der Variablen ist. Um zu verstehen, welche Typen zuweisungskompatibel sind, werden wir uns im nächsten Kapitel zuerst ansehen, was Typen überhaupt sind und welche Arten von Typen es gibt.

## 5.5 Typisierte Konstanten

 **Typisierte Konstanten** sind bei Entwicklern und Entwicklerinnen sehr beliebt, denn trotz des Namens handelt es sich nicht um Konstanten, sondern um initialisierte Variablen. Die Deklaration typisierter Konstanten sieht wie eine Mischung aus Variablen- und Konstantendeklaration aus:

```

const
    initialisierteVariable : integer = 24;

```

Initialisierte Variablen deklariert man zwar im Konstantendeklarationsblock, aber unter den unten erwähnten Bedingungen lassen sie sich trotzdem wie Variablen verwenden.

Obwohl eine automatische Initialisierung bereits viele Fehler vermeiden hilft, resultiert die Beliebtheit typisierter Konstanten aus einer anderen Eigenschaft. Lokale, d.h. in einer Prozedur oder Funktion deklarierte typisierte Konstanten

behalten nämlich ihren Wert, auch wenn man die Prozedur verlässt. In der C-Programmierung kennt man dafür die Variablenart „Static“. Diese Eigenschaft ermöglicht es, beispielsweise einen Zähler zu programmieren, der angibt, wie oft ein Ereignis durch ein Kontrollelement ausgelöst wird, und den Code in der zum Kontrollelement gehörigen Ereignisbehandlungsroutine zu plazieren.

Sie können dies testen, indem Sie ein Formular mit einem Button `btnZaehler` versehen und dessen Ereignisbehandlungsroutine zu `ONCLICK` zu folgendem Code ergänzen:

```

1 procedure TfrmEditor.btnZaehlerClick
2   (Sender: TObject);
3   const
4     Zaehler : byte = 0;
5   begin
6     Inc(Zaehler);
7     btnZaehler.caption := inttostr(Zaehler);
8   end;
```

Jedes Mal, wenn Sie auf den Button klicken, wird dessen Beschriftung um 1 erhöht. Allerdings lässt sich dieses Programm nur kompilieren, wenn die Compilerdirektive `{$J+}` eingeschaltet ist, die Sie im Register `COMPILER` des Menüs `PROJECT - OPTIONS` im Feld `ASSIGNABLE TYPED CONSTANTS` einstellen können.

Borland scheint diese Programmierpraxis aber nicht zu mögen. Dass lokale typisierte Konstanten erhalten bleiben, wird verschwiegen, und vom Gebrauch typisierter Konstanten als Variablen wird in der Online-Hilfe mit folgenden Worten ausdrücklich abgeraten:

„Die Compilerdirektive `$J` ermöglicht die Deklaration von typisierten Konstanten, die geändert werden können. Typisierte Konstanten, die im Standardzustand `{$J-}` deklariert werden, können nur gelesen und nicht verändert werden. Aus Gründen der Abwärtskompatibilität zu älteren Versionen von Delphi und Borland Pascal können typisierte Konstanten, die im Zustand `{$J+}` deklariert wurden, geändert werden. Prinzipiell handelt es sich dabei um initialisierte Variablen. **Für neue Anwendungen ist der Einsatz des Zustands `{$J+}` nicht empfehlenswert.**“

Da Borland keine Alternativen für Static-Variablen und das im Beispiel genannte Problem erwähnt, dürften typisierte Konstanten auch weiterhin beliebt bleiben.

## 5.6 Rekapitulation

### 5.6.1 Kapiteltest

- 1) Geben Sie von den folgenden Operationen an, ob sie das Resultat `true` oder `false` ergeben. Sehen Sie bei Bedarf in der Hilfe unter „precedence of operators“ nach.

`6 * 8 + 3 > 60`

`(8 > 7) and (2 > 1) xor (4 < 3)`

- 2) Geben Sie für die folgenden Operatoren an, ob sie unär oder binär sind:

`+`  
`not`  
`mod`  
`-`  
`div`

- 3) Finden Sie den Fehler im folgenden Programmcode:

```
1  const  
2      strblabla = string : 'blabla';  
3  var  
4      intHoehe : integer;
```

- 4) Richtig oder falsch?


- a) Wenn Sie versuchen, eine Variable im rechten Teil einer Zuweisung zu verwenden, bevor Sie sie mit einem Wert initialisiert haben, dann bricht die Compilierung mit einer Fehlermeldung ab.
- b) Technisch ist eine Variable eine Speicheradresse, deren Inhalt sich zur Laufzeit ändern kann.
- c) Der Wert einer typisierten Konstante kann sich zur Laufzeit nicht ändern.
- d) `const AnzSekunden = 24 * 60 * 60;` ist eine gültige Konstantendeklaration.



## 6 Typen


Typen sind Kategorisierungen von Daten. Variablen werden danach eingeteilt, ob sie Zahlen, Textteile oder andere Arten von Daten speichern. Diese Kategorisierung bestimmt sowohl, wie die Werte von Variablen gespeichert werden, als auch welche Operationen mit ihnen durchgeführt werden können.

Pascal ist eine streng typisierte Programmiersprache. Dies hat den Vorteil, dass der Compiler viele mögliche Fehler bereits als Syntaxfehler abfängt, so dass sie nicht erst zur Laufzeit auftreten.

 Dieses Kapitel führt Typen als eines der grundlegenden Konzepte von Pascal und von vielen anderen Programmiersprachen ein. Sie lernen,

- welche Arten von Typen es gibt,
- welche einfachen und strukturierten Typen in Object Pascal existieren,
- wofür man String- und Zeigertypen braucht,
- und was man bei der Kompatibilität von Datentypen beachten muss.

### 6.1 Definition und Klassifikation von Typen

 Ein **Typ** ist eine Beschreibung, wie man Daten speichern und wie man auf sie zugreifen sollte. Beachten Sie den Unterschied zu „Variable“, der tatsächlichen Speicherung von Daten. Der Typ legt den Wertebereich der Variablen fest und bestimmt die Operationen, die mit ihr ausgeführt werden können.

Pascal stellt relativ viele Datentypen zur Verfügung. Sie können aus diesen Typen aber auch Ihre eigenen Typen entwickeln. Bei den vorgegebenen Typen unterscheidet Object Pascal sechs verschiedenen Arten von Typen:

- Einfache Typen
- String-Typen
- Strukturierte Typen
- Zeigertypen

- Prozedurale Typen
- Varianttypen

Die folgende Tabelle gibt einen Überblick über die Kategorien und Typen von Delphi 2.0. Gewisse der aufgezählten Typen werden Sie nicht behandeln, sie sind trotzdem der Vollständigkeit halber aufgeführt.

<b>Einfache Typen</b>	<b>Ordinale Typen</b>	integer* shortint smallint longint byte cardinal* word boolean ByteBool WordBool LongBool char* AnsiChar WideChar <i>Aufzählungstypen</i> <i>Teilbereichstypen</i>
	<b>Realtypen</b>	real single double extended comp currency
<b>String-Typen</b>		string* (kurze oder lange Strings) AnsiString (langer String) ShortString (kurzer String)
<b>Strukturierte Typen</b>		Arraytypen Dateitypen Klassentypen Klassenreferenztypen Recordtypen Mengentypen
<b>Zeigertypen</b> <b>Prozedurale Typen</b>		Globale Prozedurzeiger Methodenzeiger
<b>Varianttypen</b>		

\* generische Typen

In den folgenden Abschnitten werden die wichtigsten dieser Typen beschrieben.

## 6.2 Einfache und String-Typen

Einfache Typen definieren geordnete Wertemengen. Es gibt zwei Basisklassen von einfachen Typen:

- Ordinale Typen
- Realtypen

☞ **Ordinale Typen** sind eine Untermenge der einfachen Typen und enthalten eine endliche Anzahl von Elementen. Alle Werte eines ordinalen Typs sind eine geordnete Menge. Jeder Wert kann über seine Indexposition innerhalb der Menge bestimmt werden. Mit Ausnahme der Werte vom Typ Integer hat das erste Element jedes ordinalen Typs die Indexposition 0, das nächste 1, und so weiter. Der Index eines Wertes vom Typ Integer ist der Wert selbst.

Da ordinale Typen eine geordnete Menge sind, bietet Pascal Standardfunktionen an, mit denen sich die Position in der Menge (`Ord`), der vorangehende (`Pred`) und der nachfolgende Wert (`Succ`) sowie der erste und der letzte Wert (`Low`, `High`) holen lassen. Weitere Informationen zu diesen Funktionen finden Sie in der Hilfe.

Wie Sie in der Tabelle bereits gesehen haben, gibt es zehn vordefinierte ordinale Typen und zwei durch die Benutzer definierbare. Von den zehn vordefinierten Typen gehören fünf zu den Integer-Typen, vier zu den Booleschen Typen. `Char` ist eine eigene Kategorie.

### 6.2.1 Integer-Typen

**Integers** sind Ganzzahlen. Object Pascal unterscheidet abhängig von Vorzeichen und Bereich die folgenden fünf fundamentalen und zwei generischen Integer-Typen (diese Unterscheidung wird in Kapitel 6.5.2 erklärt):

<b>Typ</b>	<b>Bereich</b>	<b>Format</b>
shortint	-128 .. 127	8-bit mit Vorzeichen
smallint	-32'768 .. 32'767	16-bit mit Vorzeichen
longint	-2'147'483'648 .. 2'147'483'647	32-bit mit Vorzeichen
byte	0 .. 255	8-bit ohne Vorzeichen
word	0 .. 65'535	16-bit ohne Vorzeichen
<b>generische Typen</b>		
integer	-32'768 .. 32'767 oder -2'147'483'648 .. 2'147'483'647	16-bit mit Vorzeichen 32-bit mit Vorzeichen
cardinal	0 .. 65'535 oder 0 .. 2'147'483'647	16-bit ohne Vorzeichen 32-bit ohne Vorzeichen

Sie können durch Typumwandlung Werte vom Typ Integer explizit in einen anderen Integer-Typ konvertieren.

Die Deklaration von Integer-Variablen könnte etwa so aussehen.

Var

```

    intTemperatur      : integer;
    bytZaehler         : byte;
    carBillGatesGehalt : cardinal;
```

### 6.2.2 Boolesche Typen

Werte vom **booleschen Typ** können nur einen der vordefinierten Konstantenbezeichner `False` oder `True` annehmen. `False` entspricht dabei dem Wert 0 und `True` dem Wert 1. Boolesche Typen gehören somit auch zu den ordinalen Typen. Meistens werden Boolesche Variablen bei Vergleichen (gleich, ungleich, grösser, kleiner) und bei If- und Case-Anweisungen verwendet.

Es gibt vier vordefinierte Boolesche Typen:

<b>Typ</b>	<b>Speicher</b>
boolean	1 Byte
ByteBool	1 Byte
WordBool	Zwei Bytes (ein Word)
LongBool	Vier Bytes (zwei Words)

`Boolean` ist der bevorzugte Typ und verbraucht am wenigsten Speicherplatz; `ByteBool`, `WordBool` und `LongBool`

sorgen für Kompatibilität mit anderen Sprachen und der Umgebung von Windows.

Der folgende Code gibt Ihnen ein Beispiel für Deklaration und Einsatz einer booleschen Variablen:

```
var
  bolBeendet : boolean;
...
if bolBeendet then close;
```

### 6.2.3 Der Char-Typ

☞ Der Wertebereich von Variablen des Typs **Char** besteht aus Zeichen, die nach der Sortierfolge des ANSI-Zeichensatzes angeordnet sind. Der Funktionsaufruf `Ord('a')` gibt beispielsweise die Ordinalzahl des Buchstabens 'a' zurück.

Der Char-Typ wird, wie Sie im folgenden Beispiel sehen, mit dem reservierten Wort `char` gekennzeichnet.

```
var
  chrDateiattribut : char;
```

Eine String-Konstante der Länge 1 kann auch durch einen konstanten Char-Wert dargestellt werden. Mit der Standardfunktion `Chr` kann jeder Char-Wert erzeugt werden.

Zum Thema generische und Fundamental-Char-Typen kommen wir im Unterkapitel 6.5.2.

### 6.2.4 Aufzählungs- und Teilbereichstypen

Sowohl Aufzählungs- wie Teilbereichstypen sind keine vordefinierten, sondern benutzerdefinierte Typen (siehe auch Kapitel 6.3.1).

☞ **Teilbereichstypen** sind vom Benutzer definierte Wertebereiche aus einem ordinalen Typ. Die Definition eines Teilbereichstyps spezifiziert den kleinsten und den grössten Wert im entsprechenden Teilbereich. Sie können zwar im Typendeklarationsteil explizit einen Teilbereichstyp deklarieren, wenn Sie ihn mehrmals verwenden möchten. Häufig genügt es aber, den Teilbereichstyp nur in der Variablendeklaration anzugeben.

☞ **Aufzählungstypen** definieren geordnete Wertemengen, indem einfach die Bezeichner der betreffenden Werte aufgelistet werden. Die Anordnung der Werte wird durch die angegebene Reihenfolge der Bezeichner bestimmt. Das

erste Element erhält den Wert 0, das zweite den Wert 1, und so weiter.

In Zeile 2 und 5 des Beispiels unten sehen Sie die Deklaration und anschliessende Verwendung eines Teilbereichstyps. Zeile 6 zeigt die Alternative dazu, nämlich die direkte Verwendung des Teilbereichs in der Variablendeklaration. Zeile 3 zeigt die Typendeklaration für einen Aufzählungstyp.

```

1  type
2      TWuerfelzahl = 1..6;
3      TKarten     = (Kreuz, Pik, Herz, Karo);
4  var
5      Wuerfel1    : TWuerfelzahl;
6      Wuerfel2    : 1..6;
7      KarteZiehen : TKarten;
8      bytPosition : byte;
9  begin
10     bytPosition := ord(Herz);
11 end;
```

Wenn Sie die Funktion `Ord` auf den Wert eines Aufzählungstyps anwenden, erhalten Sie dessen Position im Wertebereich. Der Variablen `bytPosition` in Zeile 10 wird also der Wert 2 zugewiesen.

#### 6.2.5 Realtypen

Ein **Realtyp** ist eine Untermenge der reellen Zahlen, die als Gleitkommazahl mit einer festen Stellenanzahl dargestellt werden können.

Es gibt fünf Arten von Realtypen. Sie unterscheiden sich in ihrem Wertebereich, der Genauigkeit der Werte und im Speicherplatzbedarf.

Typ	Bereich	Signifikante Stellen	Bedarf in Bytes
real	$2.9 * 10^{-39} .. 1.7 * 10^{38}$	11-12	6
single	$1.5 * 10^{-45} .. 3.4 * 10^{38}$	7-8	4
double	$5.0 * 10^{-324} .. 1.7 * 10^{308}$	15-16	8
extended	$3.4 * 10^{-4932} .. 1.1 * 10^{4932}$	19-20	10
comp	$-2^{63}+1 .. 2^{63}-1$	19-20	8
currency	-922'337'203'685'477.5808.. 922'337'203'685'477.5807	19-20	8

Der Typ `currency` ist ein Datentyp mit vier Nachkommastellen für Finanzberechnungen.

Der Typ `real` wird wegen der Abwärtskompatibilität mit früheren Versionen von Delphi und Borland Pascal bereitgestellt. Da das Speicherformat des Typen `Real` in den Intel-CPU's nicht vorgesehen ist, sind Operationen mit Realtypen langsamer als mit den anderen Gleitkommatypen.

#### 6.2.6 String-Typen

☞ Ein Variable des Typs **String** ist eine Zeichenfolge mit dynamischer Länge. Die Länge ist abhängig von der tatsächlichen Zeichenanzahl während der Programmausführung.

Der neue Typ `string` in Delphi 2.0 ist nicht identisch mit `string` in früheren Versionen, lässt sich aber gleich verwenden. Dieser neue String-Typ bietet drei Vorteile:

1. Die für Pascal-Strings üblichen String-Operationen lassen sich weiterhin verwenden.
2. Dadurch, dass lange Strings immer nullterminiert sind (siehe unten), hat sich die Kommunikation mit Windows-API-Funktionen erheblich vereinfacht.
3. Nur der vorhandene Speicherplatz begrenzt die Grösse dieses neuen String-Typs.

In früheren Versionen war ein String auf 255 Zeichen begrenzt. Ab Delphi 2.0 ist `string` ein generischer Typ, der abhängig von der Compilerdirektiven `{ $H }` entweder auf

☞ 255 Zeichen begrenzt ist (**kurzer String-Typ**) oder eine dynamische Länge aufweist, die maximal 2 Gigabyte betragen kann (**langer String-Typ**). Sie können also getrost Ihre Memoiren in einen String verpacken!

Wenn es erforderlich ist, in einer 32-Bit-Anwendung zwischen kurzen und langen String-Typen zu unterscheiden, verwenden Sie statt des generischen Typs `string` für kurze Strings den fundamentalen Typ `shortstring` und für lange Strings den fundamentalen Typ `ansistring`.

Intern ist eine lange String-Variable ein Zeiger auf einen dynamisch verwalteten Speicherbereich, in dem die Länge des Strings, ein Referenzzähler sowie der eigentliche String-Inhalt enthalten ist. Der Zeiger selbst ist nur vier Bytes gross. Die Verwaltung des dynamisch zugewiesenen Speichers einer langen String-Variablen geschieht automatisch und erfordert keinen zusätzlichen Code.

Der kurze String-Typ ist vor allem für die Rückwärtskompatibilität mit früheren Versionen von Delphi und Borland-Pascal vorhanden.

Es folgt ein Beispiel zu Deklaration und Anwendung des generischen Typs;

```
Var
  strKurzerString : string;
  strLangerString: string;
...
strKurzerString := 'Dieser String ist kurz';
strLangerString := 'Dieser String ist so
lang, dass er nicht in einen kurzen String
mit 255 Zeichen passt. Dieser String ist so
lang, dass er nicht in einen kurzen String
mit 255 Zeichen passt. Dieser String ist so
lang, dass er nicht in einen kurzen String
mit 255 Zeichen passt. Dieser String ist so
lang, dass er nicht in einen kurzen String
mit 255 Zeichen passt.'
```

Beachten Sie, dass `strKurzerString` unter Delphi 2.0 zwar ein kurzer String, aber vom langen String-Typ ist! Beachten Sie ausserdem, dass der lange String-Typ im allgemeinen weniger Speicherplatz beansprucht als der kurze, weil er nur Platz für die effektive Anzahl Zeichen und für die vier Bytes des Zeigers beansprucht, während ein kurzer String-Typ immer Speicherplatz für die maximale statt für die effektive Anzahl Zeichen belegt.

Damit Sie den obigen Code verwenden können, muss die Compilerdirektive `{$H+}` gesetzt sein. Dies können Sie direkt im Code angeben oder für das ganze Projekt in den Projektoptionen einstellen:



Öffnen Sie das Menü PROJECT - OPTIONS und wählen Sie das Register COMPILER. In der Gruppe SYNTAX OPTIONS finden Sie unter anderem HUGE STRINGS. Dieses Feld sollte defaultmässig angekreuzt sein. Auch den andern Feldern dieses Registers entsprechen Compilerdirektiven. Klicken Sie auf die Schaltfläche HELP, um sich die Informationen dazu anzeigen zu lassen.

Für Strings, deren maximale Länge bekannt ist, kann man den String-Typ mit einem Grössenattribut ergänzen. Die Angabe des Grössenattributs führt automatisch zu einem kurzen String-Typ, entspricht also unter Delphi 2.0 dem fundamentalen Typ `shortstring`.



Eine Ganzzahl in eckigen Klammern gibt dabei die maximale Anzahl Zeichen an. Sie haben diese Syntax bereits in der Übung auf Seite 77 verwendet:

```
strWillkommen : string[50];
```

Strings sind indexiert, so dass Sie über den Index direkt auf die einzelnen Zeichen eines Strings zugreifen können. Der Index des ersten Zeichens in einem String ist 1. Bei kurzen Strings (`shortstring` oder `string` mit `{$H-}`) enthält das Element mit Index 0 die dynamische Länge des Strings. Mit dem Pluszeichen lassen sich ausserdem längere Strings aus mehreren Teilstrings zusammensetzen (konkateneren).

In der folgenden Übung probieren Sie dies aus, indem Sie das Editor-Programm erweitern. Das Ziel ist es, eine Meldung anzuzeigen, welche die Länge und den ersten Buchstaben des Willkommenstextes anzeigt. Aus Gründen, die erst später erklärt werden, erscheint diese Meldung, bevor das Formular selbst erscheint.



Wechseln Sie in das Editor-Projekt. Suchen Sie im Quelltexteditor die bereits bearbeitete Prozedur `TfrmEditor.FormCreate`.

Ergänzen Sie die Prozedur nach der letzten Anweisungszeile `memEditor.Lines.Add(strWillkommen);` mit folgendem Code:

```
1  MessageDlg(
2      'StrWillkommen enthält '
3      + IntToStr(Length(strWillkommen))
4      + ' Zeichen' + chr(13)
5      + 'strWillkommen[0] enthält '
6      + inttostr(ord(strWillkommen[0]))
7      + chr(13) + 'Erster Buchstabe: '
8      + strWillkommen[1],
9      mtInformation, [mbOk], 0);
```

Wenn Sie das Programm nun mit F9 starten, sollten Sie folgende Anzeige erhalten:

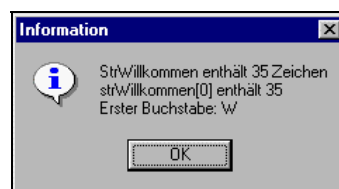


Abb. 25 Eine Übung mit Strings



`MessageDlg` ist eine Delphi-Funktion mit vier Parametern zur Anzeige von Meldungen. In Zeile 2 bis 8 übergeben Sie

den anzuzeigenden String. Mit dem Parameter `mtInformation` in Zeile 9 wählen Sie den Typ des Meldungsfensters. `[mbOk]` gibt an, dass ein OK-Button angezeigt werden soll. Der dritte Parameter legt fest, welcher Hilfebildschirm für das Meldungsfenster zur Verfügung steht. In unserem Fall ist dies keiner, deshalb steht am Ende der Zeile 9 der Wert 0.

Der String in Zeile 2 bis 8 setzt sich aus diversen Teilstrings zusammen, die mit dem Pluszeichen verbunden werden. In Zeile 3 bestimmen Sie die Länge des Strings `strWillkommen` und wandeln den erhaltenen Integer in einen String um, damit Sie ihn einbinden können. `chr(13)` in Zeile 4 steht für einen Zeilenumbruch.

Die Funktion `ord(strWillkommen[0])` in Zeile 6 liefert den gleichen Wert wie `Length`, da die aktuelle Länge eines kurzen Pascal-Strings (nicht aber eines langen Strings) in dieser Position gespeichert ist.

Zeile 8 illustriert schliesslich noch, wie man auf einzelne Zeichen des Strings zugreift.



Wenn Sie dieser Aufruf von `MessageDlg` bei zukünftigen Erweiterungen stört, dann setzen Sie die gesamte Anweisung inklusive abschliessendes Strichpunkt in geschweifte Klammern wie einen Kommentar. Diese Methode können Sie immer anwenden, wenn Sie Code behalten, aber zeitweilig ausser Kraft setzen möchten.

#### 6.2.7 Pascal-Strings und nullterminierte Strings

Falls Sie C programmiert haben, kennen Sie nullterminierte Strings. Ein **nullterminierter String** besteht aus einer Folge von Zeichen (nicht Null) gefolgt vom **Terminator-Nullzeichen** NULL (#0). Häufig wird statt des nullterminierten Strings auch ein Zeiger auf den String vom Typ `PChar` verwendet. Viele API-Funktionen von Windows benötigen beim Aufruf nullterminierte Strings als Parameter.

Vor Delphi 2.0 hat Borland für Strings ein eigenes Format verwendet, bei dem Strings eine fixe maximale Länge hatten und die effektive Länge am Anfang des Strings gespeichert war. Obwohl es in Turbo Pascal und Delphi 1.0 eingebaute Funktionen zur Umwandlung von Pascal- und nullterminierten Strings gab, war der Umgang mit diesen zwei Arten von Strings doch eher mühsam und fehleranfällig.

Mit der Einführung des generischen Typs `string` in Delphi 2.0 hat sich dies massiv vereinfacht, denn lange Strings

werden wie nullterminierte Strings automatisch durch #0 beendet. Wegen der automatischen Begrenzung durch das Nullzeichen enthält jeder lange String einen nullterminierten, und es ist somit möglich, direkt eine Typumwandlung eines langen Strings in einen PChar-Wert vorzunehmen. Die Syntax einer solchen Umwandlung lautet `PChar(S)`, wobei S ein langer String ist. Eine PChar-Typumwandlung gibt einen Zeiger auf das erste Zeichen der langen String-Variablen zurück und garantiert die Rückgabe eines Zeigers auf einen durch Null beendeten String.

### 6.3 Strukturierte Typen

- ☞ Ein **strukturierter Typ** enthält mehr als einen Wert. Die Elemente von strukturierten Typen lassen sich einzeln oder in ihrer Gesamtheit manipulieren. Sie können selber wieder strukturierte Typen sein. Es gibt keine Beschränkung hinsichtlich der Anzahl solcher geschachtelter Strukturen.

#### 6.3.1 Benutzerdefinierte Typen

Obwohl es in Delphi eine ganze Reihe vordefinierter Typen gibt, ist es für eine Programmiersprache die Möglichkeit, benutzerdefinierte Datentypen einzuführen, unbedingt nötig. In Pascal gab es die Option, selbst Typen zu deklarieren, von Anfang an.

Eigene Typendeklarationen machen Programme nicht nur lesbarer, sondern sie stellen auch sicher, dass Variablen keine unpassenden Werte zugewiesen erhalten. Wenn Sie beispielsweise Wochentage als `byte` deklarieren, müssen Sie selbst sicherstellen, dass Sie keine Werte grösser als 7 zuweisen. Wenn Sie stattdessen einen Aufzähltyp (`Mon`, `Die`, `Mit`, `Don`, `Fre`, `Sam`, `Son`) deklarieren, werden Zuweisungen von anderen Werten bereits bei der Compilierung entdeckt.

Die zwei benutzerdefinierten Typen Aufzähltyp und Teilbereichstyp, die zu den einfachen Typen gehören, haben Sie bereits kennengelernt. Alle anderen benutzerdefinierten Typen gehören zu den strukturierten Typen.

Typendeklarationsteile lassen sich im Interface- und im Implementationsteil von Units ebenso wie in Prozeduren, Funktionen und Methoden antreffen. Ein Typendeklarationsteil wird mit dem reservierten Wort `type` eingeleitet. Dann folgen die einzelnen Deklarationen, die wie in jedem Dekla-

rationsteil ein Strichpunkt abschliesst. Es ist zwar nicht zwingend, hat sich aber eingebürgert, dass man einen Typenbezeichner mit dem Grossbuchstaben T am Anfang kennzeichnet. Die allgemeine Syntax für den Typendeklarationsteil sieht also folgendermassen aus:

```
type
  TTypname1 = Typ1definition;
  TTypname2 = Typ2definition;
```

Konkrete Beispiele haben Sie bereits bei den Aufzähl- und Teilbereichstypen gesehen. In den folgenden Unterkapiteln finden Sie weitere Beispiele für strukturierte Datentypen. Für kompliziertere Typen wie Klassen müssen Sie sich bis zum Kapitel 9 gedulden.

### 6.3.2 Arraytypen

**Arrays** sind ein- oder mehrdimensionale Bereiche, die viele Variablen des gleichen Typs. enthalten. Auf jede Variable im Array lässt sich über den Namen des Arrays und den Index der Variable zugreifen, der in Klammern angegeben wird.

Um einen Arraytyp zu spezifizieren, benötigt der Compiler den ordinalen Indextyp eines Arrays, der die Anzahl der Elemente angibt, und den Basistyp. Die Anzahl der Elemente in einem Array ist das Produkt der Anzahl der Werte in jedem Indextypen. Die generelle Syntax sieht folgendermassen aus.

```
Type
  TArray = array[Indextyp1,
                 Indextyp2,...,IndextypN] of type;
```

Hier sehen Sie Beispiele für einen ein- und einen zweidimensionalen Arraytyp:

```
type
  TDateiattribut = array[1..4] of char;
  TFigur = (Koenig, Dame, Laeufer, Springer,
           Turm, Bauer);
  TSchachbrett = array[1..8,1..8] of TFigur;
var
  arrDateiattribut : TDateiattribut;
  arrMatrix : array[1..10,1..10] of string;
```

Um auf Elemente des Arrays zuzugreifen, fügen Sie dem Arraybezeichner eckige Klammern und einen oder mehrere Indexwerte hinzu. Die folgenden Zeilen weisen beispielswei-

se das dritte Element der Arrayvariablen `arrDateiattribut` und das Element `[1,1]` von `arrMatrix` zu:

```
arrDateiattribut[3] := 'h';
arrMatrix[1,1] := arrDateiattribut[3];
```

Die zweite Zuweisung ist möglich, weil ein Char intern gleich wie ein String der Länge 1 behandelt wird, und somit typenkompatibel zum Typ String ist. Auf die Zuweisungskompatibilität von Typen werden wir am Ende des Kapitels noch eingehen.

### 6.3.3 Mengentypen

☞ Ein **Mengentyp** ist eine Ansammlung von Objekten vom gleichen ordinalen Typ. Um einen Mengentyp zu deklarieren, verwenden Sie die reservierten Wörter `set of`, gefolgt vom Basistyp.

Der Wertebereich eines Mengentyps ist die Potenzmenge eines bestimmten ordinalen Typs (des Basistyps). Jeder mögliche Wert eines Mengentyp ist eine Untermenge der möglichen Werte des Basistyps.

Eine Mengentypvariable kann keinen oder alle Werte der Menge enthalten. Jeder Mengentyp kann den Wert `[]` enthalten, der leere Menge genannt wird.

Der Basistyp darf nicht mehr als 256 mögliche Werte haben. Deshalb können Sie zwar ein `set of byte`, nicht aber ein `set of integer` deklarieren. Die ordinalen Werte der Ober- und Untergrenze des Basistyps müssen im Bereich von 0 bis 255 liegen.

Mit den Operatoren `+`, `-` und `*` lassen sich Vereinigungs-, Differenz- und Teilmengen zweier Mengen bilden. Ausserdem gibt es auch Operatoren um festzustellen, ob eine Menge identisch mit oder eine Teilmenge von einer anderen Menge ist.

Im folgenden Codebeispiel sehen Sie, wie man Mengentypen deklariert und verwendet:

```
1 type
2   TLetterSet = set of 'A'..'z';
3 var
4   setLetter1, setLetter2 : TLetterSet;
5   setLetter3 : set of 'A'..'z';
6 begin
7   setLetter1 := ['a'..'c'];
8   setLetter2 := ['d'..'e'];
9   setLetter2 := setLetter2 + ['c'];
```

```

10 {Differenzmenge}
11   setLetter3 := setLetter1 - setLetter2;

```

Der von Borland missbilligte Gebrauch typisierten Konstanten als Variablen ist auch für Mengen äusserst praktisch, denn wie man im folgenden Beispiel sieht, erlaubt dies, Mengen aus nicht zusammenhängenden Teilbereichen ordinaler Typen zu bilden.

```

const
  Bezeichnerzeichen : set of '0'..'z'
    = ['0'..'9', 'A'..'Z', 'a'..'z', '_'];
begin
  if chrZeichen in Bezeichnerzeichen
    then begin ...

```

Um ein Element rasch einer Menge anzufügen oder es daraus zu entfernen, stehen die Prozeduren `Include` und `Exclude` zur Verfügung.

#### 6.3.4 Recordtypen

☞ Ein **Recordtyp** ist eine Anordnung von Feldern, die verschiedene Typen haben können.

Der Feldname und der Feldtyp müssen in der Deklaration des Recordtyps zugewiesen werden.

Der feste Teil eines Recordtyps legt die Liste der statischen Datenfelder mit jeweils einem Bezeichner und einem Typ fest. Jedes Feld enthält Informationen, die immer auf die gleiche Art und Weise gewonnen werden.

Der variante Teil eines Recordtyps verteilt den Speicherplatz für mehr als eine Liste von Feldern. Damit können Sie auf Information in mehr als einer Art zugreifen.

Die allgemeine Deklaration eines Recordtyps mit festem und variantem Teil sieht so aus:

```

1 type
2   Ttypname = record
3     {fester Teil}
4     feldname1, feldname2 : typ1;
5     feldname3 : typ2;
6     {varianter Teil}
7     case feldname4 : typ3 of
8       Wert1:
9         (feldname5a : typ2;
10          feldname6a : typ1);
11       Wert2:
12         (feldname5b : typ2);

```

```

13         Wert3:
14             (feldname5c : typ4);
15     end;

```

Das reservierte Wort `record` in Zeile 2 leitet einen Record ein. In Zeile 4 und 5 folgen die Felddeklarationen des festen Teils. Falls es einen varianten Typ gibt, folgt eine Case-Anweisung, die ein weiteres festes Feld definiert und dieses zur Unterscheidung der verschiedenen Varianten des varianten Teils benutzt. Die Case-Anweisung finden Sie in Zeile 7 und in Zeile 8 bis 14 die Feldlisten für drei verschiedene Varianten.

Jede Feldliste ist eine Variante, die den gleichen Platz im Speicher belegt. Jede Variante wird mit einer Konstanten unterschieden. Alle Konstanten müssen eindeutig und vom gleichen Typ wie das unterscheidende Feld sein.

Sie können auf den ganzen Record oder auf jedes Feld einzeln zugreifen. Um auf ein einzelnes Feld zuzugreifen, schreiben Sie den Namen des Records, einen Punkt und dann den Feldbezeichner (siehe Zeile 14 im Codebeispiel unten). Um auf mehrere Felder des Records zuzugreifen, ohne den Recordbezeichner wiederholen zu müssen, verwenden Sie `with`.

Da Records eine Art Vorstufe von Klassen und Klassen wiederum die wichtigsten Typen von Object Pascal sind, werden Sie nun Ihr Wissen über Records mit einer Übung etwas vertiefen.



Starten Sie mit FILE - NEW APPLICATION ein neues Projekt. Benennen Sie das neue Formular *frmRecords* und beschriften Sie es mit *Records*. Sichern Sie das Projekt mit FILE - SAVE ALL in einem neuen Ordner «Prog6\_1», indem Sie die Unit unter «uRecord.pas» und das Projekt unter «pRecord.dpr» abspeichern.

Fügen Sie aus dem Register STANDARD eine Komponente LABEL ein. Die Eigenschaft NAME ersetzen Sie durch *lblMeldung* und die Wertespalte neben Caption löschen Sie.

Fügen Sie aus dem Register STANDARD eine Komponente RADIOGROUP ein, benennen Sie diese *btnradgrpWoher* und beschriften Sie sie mit „Staatsbürgerschaft?“. Doppelklicken Sie in die Wertespalte neben ITEMS und tragen Sie im Editor *Schweiz* und *Ausland* in zwei verschiedenen Zeilen ein.

Wechseln Sie ins Register EVENTS und doppelklicken Sie ins Feld neben ONCLICK. Ergänzen Sie die Rumpfprozedur, so dass sie folgendermassen aussieht:



1

```
procedure TfrmRecords.btnradgroupWoherClick
```

```

2      (Sender: TObject);
3  type
4      TPerson = record
5          Name : string[40];
6          case Schweizer : boolean of
7              true  : (Heimatort : string[40]);
8              false :
9                  (Geburtsort : string[40];
10                 Land        : string[20]);
11      end;
12  var recPerson : TPerson;
13  begin
14      recPerson.Name :=
15          inputbox('Name', 'Name: ', '');
16      if btnradgrpwoher.itemindex = 0 then
17          begin recPerson.Schweizer := true;
18              recPerson.Heimatort : inputbox
19                  ('Heimatort', 'Heimatort: ', '');
20              lblMeldung.Caption := 'Sie sind ' +
21                  recPerson.Name + ' aus ' +
22                  recPerson.Heimatort + ' in der Schweiz';
23          end else begin
24              recPerson.Schweizer := false;
25              recPerson.Geburtsort := inputbox
26                  ('Geburtsort', 'Geburtsort: ', '');
27              recPerson.Land := inputbox
28                  ('Land', 'Land: ', '');
29              lblMeldung.Caption := 'Sie sind ' +
30                  recPerson.Name + ' aus '
31                  + recPerson.Geburtsort + ' ('
32                  + recPerson.Land + ')';
33          end;
34  end;

```

Führen Sie Ihr Programm mit RUN - RUN oder mit F9 aus.

Mit diesem Code haben Sie einen Record mit den zwei festen Feldern Name (Zeile 5) und Schweizer (Zeile 6) sowie den varianten Feldlisten Heimatort (Zeile 7) oder Geburtsort/Land (Zeile 9 und 10) deklariert. Mit Hilfe der Komponente TGroupbox und der Funktion inputbox haben Sie allen Feldern Werte zugewiesen. Diese Werte wurden in Zeile 30 bis 32 zu einem String zusammengesetzt und mit Hilfe des Labels lblMeldung angezeigt.

Statt einer RadioGroup-Komponente könnten Sie für eine Auswahl auch ein Listefeld in das Formular einfügen. Für die Ereignisbehandlungsroutine können Sie den bestehenden Code verwenden. Sie gehen folgendermassen vor:





Holen Sie sich aus dem Register STANDARD eine Komponente LISTBOX. Benennen Sie diese *lstxWoher*. Öffnen Sie den Stringlisteneditor mit einem Doppelklick in die Wertespalte neben der Eigenschaft ITEMS. Nun tragen Sie wieder zwei Zeilen mit dem gleichen Text, nämlich *Schweiz* und *Ausland* ein und schliessen den Stringlisteneditor. Im Register Events holen Sie sich neben ONCLICK die bereits bestehende Methode *btnradgrpWoherClick* aus der Liste. Wenn Sie jetzt das Programm mit F9 laufen lassen, können Sie die Staatsbürgerschaft entweder mit den Radio-buttons oder mit dem Listenfeld eingeben.

## 6.4 Zeigertypen



Eine **Zeigertyp**-Variable (englisch Pointer) enthält keinen Wert, sondern die Speicheradresse einer dynamischen Variablen eines Basistyps. Die dynamische Variable, auf die eine Zeigervariable zeigt, wird wie in Zeile 2 oder 8 des Codebeispiels durch Schreiben des Zeigersymbols **^** **vor der Variablen referenziert**. Um auf Werte der dynamischen Variablen zuzugreifen, muss man den Zeiger **dereferenzieren**, d.h. das Zeigersymbol **^** wie in den Zeilen 13 bis 15 **nach der Zeigervariablen** anfügen. Bevor man Zeiger dereferenzieren kann, muss man sie allerdings erst mit der Prozedur *New* erzeugen, wie dies in Zeile 11 und 12 geschieht. Nach Gebrauch oder spätestens bei Programmende sollte man den für die dynamische Variable reservierten Speicher mit der Prozedur *Dispose* wieder freigeben (siehe Zeile 18 und 19).

Typen-, Variablendeklaration und Verwendung für Zeigertypen sehen folgendermassen aus:

```

1  type
2      TptrAdresse = ^TrecAdresse;
3      TrecAdresse = record
4          strName : string;
5          strOrt  : string;
6      end;
7  var
8      ptrString : ^string;
9      ptrAdresse : TptrAdresse;
10 begin
11     new(ptrString);
12     new(ptrAdresse);
13     ptrAdresse^.strName := 'Meinname';
14     ptrString^ := 'Keine Adresse';
15     ptrAdresse^.strOrt := ptrString^;
16     lblTest.caption := ptrAdresse^.strName

```

```

17      + ', ' + ptrAdresse^.strOrt;
18      dispose(ptrAdresse);
19      dispose(ptrString);
20  end;

```

Wenn Sie Zeiger auf einen eigenen Typ erstellen wollen, dann müssen Sie wie Zeile 3 bis 6 diesen Typ innerhalb der gleichen Typendeklaration definieren.

Das reservierte Wort **nil** bezeichnet eine Zeigerkonstante, die auf nichts zeigt.

Unter Delphi 2.0 hat die Bedeutung von selbstdeklarierten Zeigern abgenommen, denn es stehen mit Objekten und den langen Strings Sprachelemente zur Verfügung, die den Speicher automatisch dynamisch verwalten.

## 6.5 Kompatibilität von Datentypen

Dass Pascal streng typisiert ist, bedeutet nicht, dass Sie Zuweisungen nur zwischen Daten vom gleichen Typ vornehmen können. Die Regeln, welche Typen einander zugewiesen werden können, fasst man unter dem Stichwort Typenkompatibilität oder Zuweisungskompatibilität zusammen. Da die neuen, generischen Typen in Delphi 2.0 im Vergleich zu den Fundamentaltypen die Regeln für die Typenkompatibilität wieder etwas vereinfachen, findet sich in diesem Unterkapitel auch ein Abschnitt über fundamentale und generische Typen.

### 6.5.1 Typen- und Zuweisungskompatibilität

Kompatibilität zwischen zwei Typen wird in Ausdrücken oder in relationalen Operationen gefordert. Die Regeln, welche Typen zueinander kompatibel sind, findet man in der Online-Hilfe unter dem Stichwort „type compatibility“ (**Kompatibilität der Datentypen**). Kompatibilität der Datentypen ist eine Voraussetzung für **Zuweisungskompatibilität**. Diese Regeln sehen Sie sich nun in der Hilfe an.



Öffnen Sie mit HELP - HELP TOPICS die Online-Hilfe, wechseln Sie ins Register INDEX, geben Sie das Stichwort *type compatibility* ein und lesen Sie die angezeigte Seite durch. Klicken Sie dann auf ASSIGNMENT COMPATIBILITY unter SEE ALSO, und lesen Sie die Regeln für die Zuweisungskompatibilität ebenfalls durch.

### 6.5.2 Fundamental- und generische Typen

Seit es Delphi für Win95 gibt, unterscheidet Object Pascals zwischen **Fundamentaltypen** und **generische Typen**.

Der Bereich und das Format von Fundamentaltypen ist von der zugrundeliegenden CPU und vom Betriebssystem unabhängig und ändert sich bei verschiedenen Implementierungen von Object Pascal nicht.

Der Bereich und das Format von generischen Typen hängt von der vorhandenen CPU und vom Betriebssystem ab.

Es gibt gegenwärtig drei Kategorien von vordefinierten Typen, die zwischen Fundamental- und generischen Typen unterscheiden:

- Integer-Typen
- Char-Typen
- String-Typen

Für alle anderen Klassen sollten Sie die vordefinierten Typen als Fundamentaltypen ansehen.

Die **generischen Integer-Typen** sind `Integer` und `Cardinal`. Der Integer-Typ stellt eine generische Ganzzahl mit Vorzeichen und der Cardinal-Typ eine generische Ganzzahl ohne Vorzeichen dar.

Die aktuellen Bereiche und Speicherformate des generischen Typs variieren bei verschiedenen Implementierungen von Object Pascal, entsprechen aber im allgemeinen denen, die aus der effektivsten Integer-Operation der zugrundeliegenden CPU und des Betriebssystems resultieren.

Object Pascal definiert zwei fundamentale Char-Typen und einen generischen Char-Typ.

Die fundamentalen Char-Typen sind:

- `AnsiChar`: nach dem erweiterten ANSI-Zeichensatz geordnet
- `WideChar`: nach dem Unicode-Zeichensatz geordnet; die ersten 256 Unicode-Zeichen entsprechen den ANSI-Zeichen

Der **generische Char-Typ** ist `Char`.

In der aktuellen Implementation von Object Pascal, entspricht `Char` dem Fundamentaltyp `AnsiChar`, aber Implementationen für andere CPUs und Betriebssysteme könnten `Char` als `WideChar` definieren.

- ☞ Der **generische String-Typ** `string` kann entweder einen langen, dynamisch zugewiesenen String (den grundlegenden ANSI-String) oder einen kurzen, statisch zugewiesenen String bezeichnen. Dies ist abhängig von der Compilerdirektiven:

```
{ $H+ } oder { $LONGSTRINGS ON }
```

- ☞ Anwendungen sollten **möglichst generische Formate verwenden**, um beste Ergebnisse mit der zugrundeliegenden CPU und dem Betriebssystem zu erzielen. Die Fundamental-Typen sollten nur verwendet werden, wenn der aktuelle Bereich und/oder das Speicherformat mit der Anwendung nicht korreliert.

Weil ein generischer Typ für mehrere fundamentale Typen steht, sind die Regeln für die Zuweisungskompatibilität von generischen Typen einfacher als jene von fundamentalen.


## 6.6 Rekapitulation

### 6.6.1 Kapiteltest

- 1) Weisen Sie den folgenden Variablen den geeigneten Typ zu. Achten Sie dabei auch auf den Speicherbedarf:
  - a) `i`: ein Zähler, um eine Schleife 12 mal zu durchlaufen
  - b) `PositionBrett`: Position einer Figur (type `TFigur`) auf einem Schachbrett (8 x 8 Felder)
  - c) `VermoegenBGates`: Vermögen von Bill Gates
- 2) Sie haben im Abschnitt über Strings zwei Arten kennengelernt, wie man die Länge eines kurzen Strings bestimmt. Finden Sie mit der Online-Hilfe die Funktion, mit der man die deklarierte Maximallänge eines kurzen Pascal-Strings erfragt.
- 3) Schauen Sie sich noch einmal das erste Codebeispiel im Abschnitt über Mengen an. Welche Elemente umfasst die Menge `setLetter3` nach der Zuweisung in der letzten Zeile?
- 4) Welche der folgenden Zuweisungen sind gültig (der Typ steht am Anfang des Bezeichners):
  - a) `stringSatz := charZeichen;`
  - b) `cardinalZahl := 64000;`  
`smallintZahl := cardinalZahl;`

- 5) Was passiert, wenn Sie den String 'Hallo Welt' einer Variablen vom Typ `string[5]` zuweisen?
- 6) Richtig oder falsch?
  - a) Auf die Felder eines Records können Sie über ihre Positionsnummer im Record zugreifen.
  - b) Eine boolesche Variable kann als Index in einem Array verwendet werden.



## 7 Ablaufsteuerung

 In diesem Kapitel lernen Sie einige einfache Sprachelemente von Object Pascal kennen, mit denen Sie den Programmablauf kontrollieren können. Sie erfahren,


- wie Anweisungen, zusammengesetzte Anweisungen, Blöcke und Gültigkeitsbereiche definiert sind,
- wie Sie die If-Anweisung verwenden,
- wie Sie die Case-Anweisung verwenden,
- und wie Sie verschiedene Arten von Schleifen in Ihre Programme einbauen können.

### 7.1 Anweisungen, zusammengesetzte Anweisungen und Blöcke

#### 7.1.1 Anweisungen

-  **Anweisungen** (statements) sind die kleinsten Einheiten eines Programms. Sie beschreiben algorithmische Aktionen, die ein Programm ausführen kann. Die einzelnen Anweisungen werden durch Strichpunkt getrennt.
-  Es gibt zwei Grundtypen von Anweisungen, nämlich einfache und strukturierte. Zu den **einfachen Anweisungen** gehören neben den Zuweisungsanweisungen, die Sie schon öfters angetroffen haben, noch Goto-Anweisungen (die Sie besser gleich wieder vergessen) und Prozeduranweisungen, mit denen eine Prozedur, Funktion oder Methode aufgerufen wird. Die **strukturierten Anweisungen** umfassen zusammengesetzte, bedingte und With-Anweisungen sowie Schleifen.

#### 7.1.2 Zusammengesetzte Anweisungen

-  **Zusammengesetzte Anweisungen** bestehen aus einer oder mehreren Anweisungen, die zwischen den reservierten Wörtern `begin` und `end` eingeschlossen sind. Die einzelnen Anweisungen werden mit einem Strichpunkt getrennt. Zusammengesetzte Anweisungen geben an, dass die enthaltenen Anweisungen in der Reihenfolge ausgeführt werden sollen, in der sie aufgeschrieben sind. Die allgemeine Syntax einer zusammengesetzten Anweisung sieht folgendermaßen aus:

```
begin
  Anweisung1;
  Anweisung2;
  ...
  AnweisungN;
end;
```

Die zusammengesetzte Anweisung wird wie eine einzige Anweisung behandelt, was an den Stellen wichtig ist, wo eine einzelne Anweisung erwartet wird, beispielsweise bei If- oder Case-Anweisungen.

### 7.1.3 Blöcke

- ☞ **Blöcke** sind nicht das gleiche wie zusammengesetzte Anweisungen. Ein Block besteht aus einem Deklarations- und einem Anweisungsteil. Der **Deklarationsteil** eines Blocks kann Labels, Konstanten-, Typen-, Variablen-, Prozedur-, Funktionsdeklarationen und Export-Anweisungen enthalten. Der **Anweisungsteil** eines Blocks ist eine zusammengesetzte Anweisung. Blöcke sind Teil der Prozedurdeklaration, Funktionsdeklaration, Methodendeklaration oder eines Programms bzw. einer Unit.

Zusammengesetzte Anweisungen bilden somit immer den Anweisungsteil eines Blockes. Darüber hinaus können sie aber auch in einzelnen Anweisungen, z.B. innerhalb von If- oder Case-Anweisungen, auftreten. Diese zwei Arten von bedingten Anweisungen werden wir uns in den nächsten zwei Unterkapiteln ansehen.

### 7.1.4 Gültigkeitsbereiche

- ☞ Blöcke sind ein wichtiges Konzept von Object Pascal, weil Blöcke die **Gültigkeitsbereiche** (englisch scopes) von Konstanten, Typen, Variablen, Prozeduren und Funktionen festlegen. Der Gültigkeitsbereich eines Bezeichners innerhalb eines Programms oder einer Unit gibt an, ob der Bezeichner von anderen Prozeduren und Funktionen im Programm oder der Unit benutzt werden kann oder nicht.
- ☞ Der Gültigkeitsbereich kann entweder **lokal** oder **global** sein. Lokale Bezeichner sind nur für die Routinen und Deklarationen sichtbar, die in dem Block enthalten sind, in welchem der Bezeichner deklariert wird.

Globale Bezeichner werden im Interface-Abschnitt der Unit deklariert und sind für alle Routinen und Deklarationen innerhalb dieser Unit sichtbar.

Beachten Sie beim Entwurf der Programmstruktur die drei Gültigkeitsbereichsregeln:

- Jeder Bezeichner hat nur in dem Block eine Bedeutung, in dem er deklariert wird. Innerhalb des Blocks erstreckt sich der Gültigkeitsbereich vom Ort der Deklaration bis zum Ende des Blocks, wobei untergeordnete Blöcke eingeschlossen sind.
- Wenn ein Bezeichner aus einem äusseren Block innerhalb eines Blocks neu definiert wird, erhält die innerste (die am tiefsten geschachtelte) Definition von der Stelle der Deklaration bis zum Ende des Blocks Vorrang.
- Wenn eine Prozedur sich selbst aufruft (Rekursion), zeigt eine Referenz einer globalen Variablen immer auf die Instanz dieser Variable im zuletzt erfolgten Prozeduraufruf.

## 7.2 Die If-Anweisung

☞ **Bedingte Anweisungen** geben Ihnen die Möglichkeit, zu kontrollieren, ob bestimmte Ausdrücke ausgewertet werden sollen oder nicht. If-Anweisungen verwenden Sie, wenn es nur zwei Möglichkeiten gibt, Case-Anweisungen dagegen, wenn es mehr als zwei Möglichkeiten gibt.

☞ Eine **If-Anweisung** besteht aus dem reservierten Wort `if`, einer Bedingung (Typ `boolean`), einer mit `then` eingeleiteten Anweisung und eventuell einer weiteren, mit `else` eingeleiteten Anweisung. Vor `else` darf in If-Anweisungen **nie** ein Strichpunkt stehen! Die Anweisung im Then-Teil wird nur ausgeführt, falls die Bedingung zutrifft. Wenn sie nicht zutrifft, wird, falls vorhanden, die Anweisung im Else-Teil ausgeführt. Sonst wird die Ausführung bei der nächsten Anweisung nach der If-Anweisung fortgesetzt.

Beispiel für eine einfache If-Anweisung:

```
If intZahl > 8
  then intZahl := 8;
intZahl2 := intZahl;
```



Beispiel für If ... Then ... Else:

```
if intZahl < 5
  then intZahl := intZahl * 3
  else intZahl := intZahl * 2;
```

Statt einer einzelnen Anweisung können im Then- und im Else-Teil auch zusammengesetzte Anweisungen der folgenden Art stehen:

```
if intZahl < 5 then begin
  intZahl2 := intZahl;
  intZahl := intZahl * 3;
end else begin
  intZahl3 := intZahl;
  intZahl := intZahl * 2;
end;
```

Then- und Else-Teil rückt man normalerweise gegenüber dem if um eine Position ein, um anzudeuten, dass diese zwei Teile dem if untergeordnet sind. Dies wird insbesondere beim Verschachteln wichtig. If-Anweisungen lassen sich nämlich über mehrere Ebenen verschachteln. Damit dies nicht zu syntaktischer Mehrdeutigkeit führt, bezieht sich else immer auf das zuletzt angegebene if, dem noch kein else zugeordnet ist. Im folgenden Codebeispiel bezieht sich deshalb das else in Zeile 5 auf das innere if in Zeile 3. Die Variable intZahl hat deshalb am Ende dieses Codes den Wert 6.

```
1  intZahl := 2;
2  if intZahl < 5 then
3    if intZahl > 3 then
4      intZahl := 2 * intZahl
5    else
6      intZahl := 3 * intZahl;
```

Wenn die Anweisung in Zeile 6 dagegen zum Else-Teil des äusseren if in Zeile 2 gehören soll, so dass intZahl den Wert 2 behält, dann gibt es drei Lösungen.

1. Man kann innere und äussere If-Anweisung vertauschen.

```
1  if intZahl > 3 then
2    if intZahl < 5
3      then intZahl := 2 * intZahl
4      else intZahl := 3 * intZahl;
```

2. Man kann unmittelbar nach else ein zweites else einfügen:

```

1  if intZahl < 5 then
2      if intZahl > 3 then
3          intZahl := 2 * intZahl
4      else
5  else
6      intZahl := 3 * intZahl;

```

3. Man kann aus dem Then-Teil des äusseren `if` eine zusammengesetzte Anweisung machen:

```

1  if intZahl < 5 then begin
2      if intZahl > 3 then
3          intZahl := 2 * intZahl;
4  end else
5      intZahl := 3 * intZahl;

```

Man kann nicht nur `If`-Anweisungen, sondern auch die Bedingungen verschachteln. Dafür stehen die logischen Operatoren `and`, `or`, `xor` und `not` sowie Klammern zur Verfügung. Zur Auswertung solch verschachtelter Bedingungen wird die boolesche Logik verwendet.

Logischer Operator	Art	Boolesche Logik
<b>and</b>	binär	ergibt <code>true</code> , wenn beide Operanden <code>true</code> ergeben
<b>or</b>	binär	ergibt <code>true</code> , wenn mindestens ein Operand <code>true</code> ergibt
<b>xor</b>	binär	ergibt <code>true</code> , wenn genau ein Operand <code>true</code> ergibt
<b>not</b>	unär	invertiert den Wert des Operanden

Das folgende Codebeispiel verdoppelt beispielsweise 4 und 5 und verdreifacht alle anderen Zahlen.

```

if (intZahl > 3) and not(intZahl >= 6)
    then intZahl := 2 * intZahl
    else intZahl := 3 * intZahl;

```

Mit etwas gedanklicher Vorarbeit lassen sich mit diesen Konstrukten bedingte Anweisungen beliebiger Komplexität realisieren.

### 7.3 Die Case-Anweisung

☞ Eine **Case-Anweisung** ist eine bedingte Anweisung, die sich besonders für Situationen mit mehr als zwei Alternativen eignet. Eine Case-Anweisung haben Sie im varianten Teil eines Records bereits kennengelernt.

Die Case-Anweisung (Fallunterscheidung) besteht aus einem Ausdruck (dem Selektor) und einer Liste von Anweisungen. Vor jeder Anweisung stehen ein oder mehrere


Case-Konstanten, ein Case-Bereich oder das reservierte Wort `else`. Nach jeder Anweisung steht wie immer ein Strichpunkt. Die allgemeine Syntax sieht folgendermassen aus:

```
Case Selektor of  
    Konstante1 oder Konstantenbereich1 :  
        Anweisung1;  
    Konstante2 oder Konstantenbereich2 :  
        Anweisung2;  
    ...  
    KonstanteN oder KonstantenbereichN :  
        AnweisungN;  
end;
```

Die Case-Konstanten müssen eindeutig sein, die Bereiche innerhalb der Case-Anweisungen dürfen sich also **nicht überlappen**. Der Typ der Case-Konstanten muss ein ordinaler Typ und mit dem Typ des Selektors kompatibel sein. Es ist zwar nicht zwingend, führt aber zu schnelleren Programmen, wenn Sie die Case-Konstanten in aufsteigender Reihenfolge angeben.


Die Case-Anweisung führt genau eine mit einer Case-Konstanten markierte Anweisung aus, und zwar diejenige, deren Case-Konstante mit dem Wert des Selektors übereinstimmt. Statt einer einzelnen Anweisung kann nach der Konstanten wie im Then- und Else-Teil einer If-Anweisung eine zusammengesetzte Anweisung stehen. Im Gegensatz zur If-Anweisungen ist es deshalb bei einer Case-Anweisung möglich, dass vor einem `else` ein Strichpunkt steht.

Für eine Zahl vom Typ `Cardinal` lässt sich das erste Codebeispiel mit den verschachtelten If-Anweisungen (Seite 105) auch als Case-Anweisung der folgenden Art formulieren:

```
 1 Case carZahl of  
2   0..3 : carZahl := carZahl;  
3   4    : carZahl := 2 * carZahl;  
4   else carZahl := 3 * carZahl;  
5 end;
```

Wenn es möglich ist, verschachtelte If-Anweisungen als Case-Anweisungen zu formulieren, dann ist letzteres vorzuziehen, weil es den Code lesbarer macht.

## 7.4 Schleifen-Programmierung

 **Schleifen** (englisch loops) ermöglichen es, eine oder mehrere Anweisungen wiederholt auszuführen, solange oder bis eine Bedingung erfüllt ist.


Object Pascal bietet Ihnen drei Arten von Schleifen, aus denen Sie je nach Situation auswählen können:

Anweisung	Situation
<code>For..to / downto..do</code>	Es gibt eine feste Anzahl Schleifendurchläufe.
<code>While..do</code>	Die Bedingung wird vor Eintritt in die Schleife geprüft, so dass die Schleife eventuell nicht durchlaufen wird.
<code>Repeat..until</code>	Die Schleife wird mindestens einmal durchlaufen, bevor die Bedingung geprüft wird.

Welche Art Sie wählen, hängt davon ab, welche Aktionen Sie in der Schleife ausführen möchten, und wieviel Sie über die Bedingungen vor Eintritt in die Schleife wissen.

Bei While- und Repeat-Schleifen müssen die Variablen der Bedingung im Innern der Schleife verändert werden, so dass es nicht zu einer Endlosschleife kommt. Falls Sie beim Test in einer Endlosschleife hängen geblieben sind, dann hilft unter Windows 95 das gleichzeitige Drücken von CTRL - ALT - DEL und anschliessend ein Klick auf TASK BEENDEN.

### 7.4.1 Die For-Schleife

 Eine **For-Schleife** führt eine durch eine Laufvariable festgesetzte Anzahl Schleifendurchläufe aus. Hier die allgemeine Syntax:

```
For Laufvariable :=
    Anfangswert to/downto Endwert do
    einfache_oder_zusammengesetzte Anweisung;
```

For-Schleifen sind sinnvoll, wenn Sie zu Beginn genau wissen, wie viele Male die Schleife durchlaufen werden soll. Die Laufvariable beginnt immer beim Anfangswert und endet beim Endwert.

Sie werden nun eine weitere If-Anweisung und eine For-Schleife in Ihr Editor-Programm einbauen:



Öffnen Sie das Editor-Projekt und fügen Sie einen weiteren Button in Ihr Hauptformular ein. Nennen Sie diesen Button *btnZeilenNr* und beschriften Sie ihn mit *Zeilennummern anzeigen*.

Verlängern Sie den Button, bis die ganze Beschriftung sichtbar ist.

Ergänzen Sie die Ereignisbehandlungsprozedur für ONCLICK dieses Buttons mit folgender If-Anweisung:

```

1  If btnZeilenNr.caption = 'Zeilennummern
2  anzeigen'
3      then begin
4          btnZeilenNr.caption :=
5              'Zeilennummern löschen';
6      end else begin
7          btnZeilenNr.caption :=
8              'Zeilennummern anzeigen';
9      end;
```

Setzen Sie die Eigenschaft `VISIBLE` des neuen Buttons auf *false*. Suchen Sie im Quelltexteditor die Prozedur `TfrmEditor.btnOpenClick` und fügen Sie im Then-Teil von deren If-Anweisung folgende Anweisung an:

```
btnZeilenNr.Visible := true;
```

Führen Sie das Programm in diesem Zustand aus, öffnen Sie eine Datei und testen Sie, was die If-Anweisung ihres neuen Buttons bewirkt.

Wenn Sie alles richtig gemacht haben, ist der neue Knopf so lange unsichtbar, bis Sie eine Datei öffnen. Wenn Sie ihn nun mehrmals anklicken, wechselt die Beschriftung. Der Zweck dieses Knopfes ist es, vorne an jede Zeile eine Zeilennummer anzufügen. Dazu verwenden Sie einen Zähler und eine For-Schleife. Fahren Sie folgendermassen fort:



Gehen Sie zurück zu der neuen Prozedur `TfrmEditor.btnZeilenNrClick` und fügen Sie vor `begin` folgenden zwei Variablendeklaration ein:

```

var
  carAnzZeilen : cardinal;
  carPosSpace : cardinal;
```

In den Then-Teil der If-Anweisung `btnZeilenNr.caption = 'Zei...` fügen Sie eine For-Schleife ein, um die Zeilennummer vorne an jede Zeile zu hängen:

```

for carAnzZeilen := 0 to memEditor.Lines.Count - 1 do begin
  memEditor.Lines[carAnzZeilen] :=
    inttostr(carAnzZeilen + 1) + ' ' +
    memEditor.Lines[carAnzZeilen];
end;
```

Die einzelnen Zeilen einer Memo-Komponente sind Strings, auf die man mit `memEditor.Lines[carAnzZeilen]` ein-

zeln zugreifen kann. Die Anzahl Zeilen kann man mit der Methode `Count` herausfinden. Die erste Zeile hat den Index 0, deshalb läuft die For-Schleife von 0 bis `Lines.Count - 1`. Damit die Zeilennummer in den String eingefügt werden kann, muss man sie zuerst mit der Funktion `inttostr` in den kompatiblen String-Typ umwandeln.

Nun werden Sie noch eine zweite For-Schleife im Else-Teil einfügen, um die Zeilennummern wieder zu löschen. Dazu benötigen Sie `Pos()` und `Copy()`, zwei Funktionen speziell für die Manipulation von Strings. `Pos` gibt Ihnen die Position eines Teilstrings in einem String und `Copy` liefert einen Teilstring einer bestimmten Länge aus einem String. Mit diesen zwei Funktionen schneiden Sie den vorderen Teil der Zeile inklusive Leerschlag ab. Die genaue Syntax der zwei Funktionen finden Sie in der Hilfe.



Geben Sie diese zweite For-Anweisung im Else-Teil ein:

```
for carAnzZeilen := 0 to memEditor.Lines.Count - 1 do begin
  carPosSpace := pos(' ', memEditor.Lines[carAnzZeilen]);
  memEditor.Lines[carAnzZeilen] :=
    copy(memEditor.Lines[carAnzZeilen],
      carPosSpace + 1,
      Length(memEditor.Lines[carAnzZeilen]) - carPosSpace);
end;
```

Kompilieren und testen Sie das Programm.


#### 7.4.2 Die While-Schleife



Wie Sie bereits erfahren haben, steht bei der **While-Schleife** die Bedingung am Anfang. Wenn sie bereits das erste Mal nicht erfüllt ist, wird diese Schleife gar nie durchlaufen. Sonst durchläuft das Programm die Schleife so lange, bis die Bedingung erfüllt ist. Die allgemeine Syntax sieht so aus:

```
While Bedingung = true do
  einfache_oder_zusammengesetzte_Anweisung;
```

Ein konkretes Beispiel könnte so aussehen:

```
 1 carZahl := 2;
  2 While carZahl < 256 do begin
  3   carZahl := carZahl * carZahl;
  4   lblWhile.caption := inttostr(carZahl);
  5 end;
```

### 7.4.3 Die Repeat-Schleife

- ☞ Eine **Repeat-Schleife** wird solange durchlaufen, bis die Abbruchbedingung am Ende erfüllt ist, mindestens aber ein Mal. Die allgemeine Syntax ist ziemlich einfach:

```
Repeat
    Anweisung1;
    Anweisung2;
    ...
    AnweisungN;
until Bedingung = true;
```

Beachten Sie, dass es hier im Unterschied zu For- und While-Schleifen nicht notwendig ist, mehrere Anweisungen in der Schleife mit `begin` und `end` zu einer zusammengesetzten Anweisung zu verbinden, denn `repeat` und `until` bilden bereits eine Klammer um die Anweisungen.

Unsere While-Schleife von vorhin lässt sich auch als Repeat-Schleife formulieren:

```
☐ 1 carZahl := 2;
   2 Repeat
   3     carZahl := carZahl * carZahl;
   4     lblWhile.caption := inttostr(carZahl);
   5 until carZahl >= 256;
```


Diese Repeat-Schleife liefert genau das gleiche Resultat wie die While-Schleife, solange wir die Variable `carZahl` in der ersten Zeile nicht auf den Wert 256 setzen. In diesem Fall wird die While-Schleife nicht durchlaufen und folglich auch kein Wert angezeigt, die Repeat-Schleife zeigt dagegen den Wert 65'536 an.

## 7.5 Rekapitulation

### 7.5.1 Übung7\_1

Schreiben Sie mit möglichst wenig Zeilen Code ein Programm, das eine Listbox mit den Grossbuchstaben des Alphabets füllt und wenden Sie dazu Ihr neu erworbenes Wissen über Schleifen an.

## 8 Prozeduren und Funktionen


 Eines der wichtigsten Mittel, um komplexe Programme in kleinere, übersichtliche und wiederverwendbare Einheiten zu zerlegen, sind Prozeduren und Funktionen. Sie lernen in diesem Kapitel,

- was Prozeduren und Funktionen sind und weshalb man sie verwendet,
- was Parameter sind, welche verschiedenen Arten es gibt und wie man sie verwendet, um Daten in Prozeduren und Funktionen zu übergeben,
- wie und weshalb Sie lokale Deklarationsteile verwenden sollten,
- was Prozeduren von Funktionen unterscheidet,
- und schliesslich, wie Prozeduren und Funktionen implementiert, aufgerufen und deklariert werden.

Der Aufbau des Kapitels geht vom Teil zu Ganzen. Sie lernen also zuerst Parameterlisten und lokale Deklarationsteile kennen, denn die Syntax dieser zwei Elemente ist für Prozeduren und Funktionen gleich. Erst wenn Sie die Bausteine dieser Routinen kennen, werden Sie in deren Implementierung und Aufruf eingeführt.

### 8.1 Einführung

Anspruchsvollere Windows-Applikationen bestehen meist aus mehreren Tausend Zeilen Programmcode. Um bei der Entwicklung den Überblick zu behalten, empfiehlt es sich, ein Programm in kleinere, voneinander relativ unabhängige Einheiten aufzuteilen. Object Pascal bietet dafür diverse Sprachelemente an. Bereits kennengelernt haben Sie Units, Blöcke und zusammengesetzte Anweisungen.

 Eines der mächtigsten Sprachkonstrukte, um Ordnung und Übersicht in Pascal-Code zu bringen, sind **Routinen**, manchmal auch **Subroutinen** genannt. Unter diesem Begriff fasst man Prozeduren, Funktionen und Methoden zusammen. Routinen bestehen aus einem **Routinenkopf**, einem **optionalen lokalen Deklarationsteil** und einem **Anweisungsteil**. Damit hat man eine Routine definiert, in Aktion tritt sie aber erst durch einen **Routinenaufruf**. Wenn man eine Routine auch anderen als jener Unit zugänglich machen möchte, in der die Routine steht, dann muss der Rou-



tinienkopf ausserdem als **Routinendeklaration im Interface-Teil der Unit** eingefügt werden.

☞ Routinen ermöglichen eine Programmierpraxis, die man **information hiding** nennt. Mehrere Anweisungen, die eine gemeinsame, häufig wiederkehrende Aufgabe erfüllen, werden in einer Routine zusammengeschlossen. Um diese Anweisungen zu durchlaufen, genügt fortan ein Routinenaufruf. Die Details, wie die Routine diese Aufgabe erfüllt, werden vor der aufrufenden Anweisung verborgen.

Delphi stellt in der Run Time Library (RTL) eine ganze Reihe vordefinierter Prozeduren und Funktionen zur Verfügung. Die RTL ist ein gutes Beispiel für information hiding, denn Sie können mit diesen Routinen problemlos arbeiten, ohne deren innere Struktur zu kennen. Daneben können Sie Ihre eigenen Routinen entwickeln. Eine bestimmte Aufgabe eignet sich dann dafür, in eine Routine ausgelagert zu werden, wenn sie häufig wiederkehrt und wenn sie mit dem restlichen Code nur über wenige Daten verbunden ist.

Prozeduren und Funktionen haben zwar etwas an Bedeutung verloren, seit es objektorientierte Programmierung und Methoden gibt, aber das meiste, was Sie in diesem Kapitel lernen, lässt sich, wie Sie im nächsten Kapitel sehen werden, direkt auf Methoden von Objekten anwenden.

## 8.2 Parameter

Gute Prozeduren und Funktionen sind vom übrigen Code möglichst unabhängig. Dennoch bezieht fast jede Prozedur oder Funktion gewisse Werte von aussen und ändert Variablen des restlichen Programms. Die Prinzipien der modularen Programmierung verlangen, dass alle diese externen Werte und Variablen als **Parameter einer Schnittstelle** in die Routine übergeben werden, so dass die Parameterliste im Routinenkopf einen vollständigen Überblick über den Austausch von Daten zwischen Programm und Routine liefert. Ein solches Vorgehen ist eine wichtige Vorbedingung dafür, dass mehrere Personen speditiv am gleichen Projekt arbeiten können, und es erhöht die Wartbarkeit von Programmen.

☞ **Parameter** übergeben Daten, das heisst Werte, Konstanten und Variablen, an Prozeduren oder Funktionen und liefern veränderte Variablenwerte aus den Routinen zurück an den aufrufenden Code.

- ☞ Die Implementierung einer Prozedur oder Funktion legt die **Liste der formalen Parameter** fest. Der Prozedur- oder Funktionsaufruf beinhaltet dann die **Liste der aktuellen Parameter**, die in Zahl und Typ den formalen Parametern entsprechen müssen.

Jeder Parameter, der in einer Liste der formalen Parameter deklariert ist, ist lokal zu der Prozedur oder Funktion in der er deklariert wird. Auf die Parameter kann innerhalb des zu der Prozedur oder Funktion gehörenden Blocks zugegriffen werden.

Es gibt fünf Arten von Parametern:

1. **Werteparameter** (call by value): Formale Werteparameter bestehen aus einem Bezeichner, gefolgt vom Typ, aber ohne vorangehendes `var`. Werteparameter lassen sich innerhalb der Routine ändern, ohne dass der Wert des aktuellen Parameters ausserhalb der Routine tangiert wird.
2. **Konstante Parameter**: Formale konstante Parameter bestehen aus dem reservierten Wort `const`, gefolgt vom Bezeichner und vom Typ. Konstante Parameter sind im Gegensatz zu Werteparametern auch innerhalb der Routine konstant, erlauben also keine Zuweisungen. Verwenden Sie konstante Parameter anstelle von Werteparametern, wenn Sie verhindern möchten, dass ein formaler Parameter während der Ausführung einer Routine seinen Wert ändert. Intern werden konstante Parameter wie Variablenparameter per Referenz übergeben. Dies spart bei grossen Variablen Speicherplatz.
3. **Variablenparameter** (call by reference): Ein Variablenparameter übergibt eine Variable per Referenz an die Prozedur oder Funktion. Das bedeutet, dass die Adresse des Parameters übergeben wird und so der Wert des Parameters verwendet und verändert werden kann. Für Variablenparameter wird in der formalen Parameterliste das reservierte Wort `var` vor Bezeichner und Typ gesetzt. Variablenparameter verwendet man für alle Daten, die eine Prozedur verändert an den aufrufenden Code zurückliefern soll. Funktionen können mit Variablenparametern weitere Daten ausser dem Funktionswert zurückliefern.
4. **Nicht typisierte Parameter**: Formale nicht typisierte Parameter sind konstante oder Variablenparameter ohne Typ-Angabe. Damit kann der aktuelle Parameter beim Routinenaufruf jede Variablen- oder Konstantenreferenz beliebigen Typs sein.

**5. Offene Array-Parameter:** Mit offenen Array-Parametern ist es möglich, Arrays verschiedener Grösse an eine Prozedur oder Funktion zu übergeben.

Weitaus die wichtigsten dieser fünf sind Variablen- und Werteparameter.

Formale Parameterlisten stehen im Kopf einer Routine zwischen runden Klammern. Die einzelnen Parameterdeklarationen sind durch Strichpunkt getrennt. Eine einzelne Parameterdeklaration hat die Syntax:

```
optional_reserviertes_Wort Bezeichner : Typ;
```

Mehrere Parameter der gleichen Art und des gleichen Typs lassen sich in der gleichen Parameterdeklaration zusammenfassen. Ihre Bezeichner werden dann mit Kommas getrennt. Es folgen ein paar Beispiele für formale Parameterlisten.

Werte-, Variablen- und konstante Parameter:

```
(intZahl1, intZahl2 : Integer; var strS1 : string; const intZahl3 : integer)
```

Offener Array-Parameter:

```
(var S: array of Char)
```

Nicht typisierte Parameter:

```
(var X, Y)
```

### 8.3 Lokale Deklarationsteile

Bereits das Unterkapitel über Gültigkeitsbereiche auf Seite 103 hat Sie damit vertraut gemacht, dass es nicht sinnvoll ist, in einem Programm alle Daten und Routinen allen Anweisungen zugänglich zu machen. Stattdessen gibt es globale und lokale Gültigkeitsbereiche. Die Konstanten, Typen, Variablen und Routinen eines Deklarationsteils sind nur in dessen Gültigkeitsbereich zugänglich.

Routinen bilden nun eigene lokale Gültigkeitsbereiche. Prozeduren und Funktionen können einen eigenen Deklarationsteil aufweisen, in dem wie im Deklarationsteil einer Unit lokale Konstanten, Typen, Variablen und Routinen deklariert werden. Es ist allerdings zu beachten, dass **Klassentypen** nicht in Prozeduren und Funktionen, sondern immer nur in Units und Programmen deklariert werden können.



Ein lokaler Deklarationsteil steht zwischen Routinenkopf und dem vom reservierten Wort **begin** eingeleiteten Anweisungsteil. Die Syntax eines lokalen Deklarationsteils entspricht genau jener einer Unit, d.h. die einzelnen Teile werden mit **const**, **type**, **var** eingeleitet und mit den einzelnen Deklarationen fortgesetzt:

```
Routinenkopf;
const
    lokale_Const1 = Wert 1;
type
    lokaler_typ1 = Typ1;
var
    lokale_var1 : lokaler_typ1;
    lokale_var2 : integer;

    procedure lokale_Prozedur;
    Deklarationsteil;
    begin {Anweisungsteil lokale Prozedur}
        Anweisung;
    end;

begin {Anweisungsteil Routine}
```

Diese lokalen Bezeichner sind nur innerhalb der Routine verfügbar und werden bei Verlassen der Prozedur oder Funktion aus dem Arbeitsspeicher gelöscht.

Da in grossen Programmen Bezeichner durchaus mehrfach vorkommen können, gibt es eine Hierarchie: Der lokalste Bezeichner hat dabei immer Vorrang vor allen anderen. Wenn Sie also eine globale Variable *i* vom Typ Integer deklariert haben und den Bezeichner *i* innerhalb einer Prozedur für eine andere Variable *i* verwenden, dann ist die globale Variable *i* in dieser Prozedur nicht verfügbar, weil dieser Bezeichner von der lokalen Variablen beansprucht wird. Die globale Variable wird von den Aktionen der Prozedur aber auch nicht tangiert und steht nach Verlassen der Prozedur wieder unverändert zur Verfügung.

☞ Eine Rolle spielt dies insofern, weil es in Pascal leider möglich ist, in Routinen auch auf die Daten der umgebenden Blöcke zuzugreifen, ohne dass sie in der Parameterliste übergeben worden sind. Dies ist aber schlechte Programmierpraxis und läuft dem Sinn und Zweck von Routinen diametral entgegen. Sie sollten deshalb auf diese Möglichkeit völlig verzichten und in Routinen nur lokale oder in der Parameterliste übergebene Konstanten, Variablen und Typen verwenden.

## 8.4 Prozeduren und Funktionen

Prozeduren und Funktionen haben eine sehr ähnliche Struktur: Sie weisen beide einen Kopf mit Parameterliste, einen optionalen Deklarations- und einen Anweisungsteil auf.

### 8.4.1 Unterschiede zwischen Prozeduren und Funktionen

Funktionen unterscheiden sich von Prozeduren dadurch, dass sie direkt einen Wert, den **Funktionswert** zurückliefern. Prozeduren tauschen dagegen mit dem restlichen Code Daten nur indirekt aus, das heisst über die Parameterliste. Funktionsaufrufe lassen sich deshalb als Teile von Anweisungen einsetzen, zum Beispiel im rechten Teil einer Zuweisungsanweisung oder als Werteparameter in einem Prozeduraufruf. Prozeduraufrufe sind dagegen immer eigenständige Anweisungen.

### 8.4.2 Implementierung von Prozeduren

Die Implementierung einer Prozedur verbindet einen Bezeichner und einen Anweisungsblock zu einer Prozedur. Die generelle Syntax sieht folgendermassen aus:

```
{Kopf}  
Procedure Prozedurname (Parameterliste);  
  {Deklarationsteil}  
  const  
    const1 := Wert1;  
  var  
    var1 : Typ1;  
    {weitere lokale Deklarationen}  
  {Anweisungsteil}  
begin  
  Anweisung1;  
  Anweisung2;  
  ...  
end;
```

Als konkretes Beispiel sehen Sie hier eine Prozedur, die ein Datum initialisiert, indem sie die Variablenparameter für Jahr, Monat und Tag setzt:

```

procedure Init_Datum
  (var Jahr, Monat, Tag : integer);
begin
  Jahr := 60;
  Monat := 1;
  Tag := 1;
end;

```

#### 8.4.3 Implementierung von Funktionen

Das reservierte Wort `Function` definiert einen Programmteil, der einen Wert berechnet und als Funktionsergebnis zurückgibt. Der Kopf der Funktion legt den Bezeichner, die formalen Parameter (soweit vorhanden) und den Ergebnistyp der Funktion fest. Funktionen können Werte jeden Typs ausser Dateitypen zurückgeben.

Eine Funktion kann nach dem Funktionskopf einen Deklarationsteil enthalten. Abgesehen vom Funktionskopf und von jener Anweisung, die den Funktionswert zuweist, ist die Syntax einer Funktion mit jener einer Prozedur identisch:

```

{Kopf}
Function Funktionsname (Parameterliste)
  : Typ_des_Funktionswerts;
  {Deklarationsteil}
  const
    const1 := Wert1;
  var
    var1 : Typ1;
    {weitere lokale Deklarationen}
  {Anweisungsteil}
begin
  Anweisung1;
  Anweisung2;
  ...
  Result := Ausdruck;
  ...
end;

```

Anstelle der impliziten Variablen `result` kann auch der Funktionsname in der Zuweisungsanweisung erscheinen. Das folgende Beispiel verdoppelt eine Zahl:

```
Function Verdoppeln
  (intZahl : integer) : integer;
begin
  result := 2 * intZahl;
end;
```

Ein weiteres konkretes Beispiel für eine Funktion finden Sie in der Übung im Abschnitt 8.4.6.

#### 8.4.4 Prozedur- und Funktionsaufrufe

Die syntaktischen Unterschiede zwischen Prozeduren und Funktionen sind zwar gering, aber für den Aufruf spielen sie eine Rolle. Ein Prozeduraufruf ist eine eigene Anweisung, ein Funktionsaufruf dagegen ein Teil einer anderen Anweisung, z.B. einer Zuweisungsanweisung oder eines Prozeduraufrufs. Hier die Syntax eines Prozeduraufrufs:

```
Prozedur-Bezeichner(parameter1, parameter2);
```

Im Gegensatz zu den formalen Parametern sind die aktuellen Parameter zwischen den Klammern durch Kommas getrennt und weisen keine Typen mehr auf.

Hier der Aufruf für unser Prozedurbeispiel von vorhin:

```
Init_Datum(intJahr, intMonat, intTag);
```

Abgesehen davon, dass sie in eine Anweisung eingebettet sind, sehen Funktionsaufrufe genau gleich aus:

```
var1 := Funktions-Bezeichner(param1,param2);
```

Unser Funktionsbeispiel von vorhin ruft man folgendermaßen auf:

```
intZahl1 := Verdoppeln(intZahl2);
```

Weil Funktionen auch Teil von Funktions- und Prozeduraufrufen sein können, sind die folgenden verschachtelten Aufrufe ebenfalls gültig:

```
lblAnzeige.caption
  := inttostr(Verdoppeln(intZahl2));
intZahl1 :=Verdoppeln(Verdoppeln(intZahl2));
```

#### 8.4.5 Deklaration im Interface-Teil

Prozeduren und Funktionen sind, wie Sie bereits gehört haben, im Normalfall nur in der Unit gültig, wo sie implementiert sind. Wenn Sie sie anderen Units zugänglich machen möchten, dann müssen Sie im Interface-Teil der Ursprungs-Unit eine Funktions- oder Prozedurdeklaration einfügen. Ei-

ne solche Deklaration besteht in der Wiederholung des Routinenkopfs mit seiner formalen Parameterliste an einem beliebigen Ort im Interface-Teil.

Allgemeine Syntax:

```
Procedure procName(formale_Parameterliste);
```

Beispiel für Prozedurdeklaration im Interface-Teil:

```
procedure Init_Datum
  (var Jahr, Monat, Tag : integer);
```

Beispiel für Funktionsdeklaration im Interface-Teil:

```
Function Verdoppeln(
  intZahl : integer) : integer;
```

#### 8.4.6 Eine eigene Funktion erstellen

Nun haben Sie das Rüstzeug zusammen, um selbst eine Funktion zu schreiben. Wir machen im Editor-Projekt bei jener Methode weiter, die Sie in Kapitel 7 geschrieben haben. Sie lernen in dieser Übung nicht nur, wie man eine eigene Funktion deklariert, sondern auch, wie man sie aufruft und anderen Units verfügbar macht.

Sie erinnern sich: Sie haben einen Button erstellt, mit dem sich bei einer geladenen Datei Zeilennummern anfügen oder entfernen lassen. Die zweite Aufgabe war etwas schwieriger, ging es doch darum, den Leerschlag zwischen Zeilennummer und Text zu suchen und den restlichen String zurückzuliefern. Dies ist eine Aufgabe, die sich ausgezeichnet für eine Funktion eignet. Damit eine solche Funktion in vielen Situationen verwendbar ist, beschränken wir uns beim gesuchten Zeichen nicht auf einen Leerschlag, sondern lassen beliebige Zeichen vom Typ `char` zu.

Wir erstellen also unter dem Bezeichner *StringNachZeichen* eine Funktion, der ein String und ein beliebiges Zeichen als Werteparameter übergeben werden, und dessen Funktionswert den Reststring nach dem Zeichen zurückliefert.



Öffnen Sie das Editor-Projekt und suchen Sie im Quelltexteditor die Methode `TfrmEditor.btnZeilenNrClick`, an der Sie im Kapitel 7 gearbeitet haben.

Plazieren Sie die folgende Funktion unmittelbar vor dieser Methode. Sie müssen die Funktion vor die Methode setzen, weil Sie sie in der Methode aufrufen möchten und Routinen nur vom Ort ihrer Deklaration bis zum Ende Blocks gültig sind.



```
1 function StringNachZeichen(strQuelle :
2   string; chrZeichen : char) : string;
```



```

3  var
4      carPosSpace  : cardinal;
5  begin
6      carPosSpace := pos(chrZeichen, strQuelle);
7      result := copy(strQuelle, carPosSpace + 1,
8          Length(strQuelle) - carPosSpace);
9  end;

```

Nun können Sie die Methode `TfrmEditor.btnZeilenNrClick` überarbeiten. Die lokale Variable `carPosSpace` haben Sie in die Funktion ausgelagert, so dass sie in der Methode nicht mehr länger benötigt wird. Löschen Sie also deren Deklaration in der Methode. Wenn Sie die alten Teile dieser Methode nicht löschen möchten, dann können Sie sie stattdessen als Kommentare in geschweifte Klammern setzen.

Im Anweisungsteil der Methode steht eine If-Anweisung, deren Else-Teil in einer For-Schleife die Zeilennummern entfernt. Den ganzen Code innerhalb der For-Schleife werden Sie nun durch den folgenden Funktionsaufruf ersetzen:

```

memEditor.Lines[carAnzZeilen] :=
    StringNachZeichen(
        memEditor.Lines[carAnzZeilen], ' ');

```

Da Sie diese Funktion vielleicht später in anderen Units benutzen möchten, setzen Sie nun noch eine Funktionsdeklaration in den Interface-Teil der Unit: Markieren Sie den ganzen Funktionskopf bis zum Strichpunkt und kopieren Sie ihn mit CTRL C.

Suchen Sie im oberen Teil der Unit das reservierte Wort `implementation` und fügen Sie mit CTRL V den Funktionskopf eine Zeile vorher ein. Damit ist die Funktion für andere Units verfügbar, wenn Sie den Namen der Unit «uMainEdi» in deren Uses-Klausel aufnehmen.

Lassen Sie Ihr Programm laufen und testen Sie, ob die Zeilennummern immer noch richtig eingefügt und entfernt werden.

Obwohl die folgende Übung keine unmittelbare Anwendung zu Funktionen und Prozeduren ist, werden Sie nun das Editor-Projekt weiter ausbauen, indem Sie den Editor mit einer Menüleiste versehen. Eine Menüleiste benötigen wir für die Übungen im nächsten Kapitel. Ausserdem ist Ihnen bestimmt schon aufgefallen, dass sich zwar der Editor vergrössern lässt, nicht aber das Memofeld, in dem der Text steht. Mit einem Panel können Sie diesen Schönheitsfehler beheben.



Vergrössern Sie im Editor-Projekt das Hauptformular gegen unten, so dass unterhalb der Schaltflächen ein leerer Streifen zum Vorschein kommt, der etwas höher als die Schaltflächen ist. Setzen Sie eine Panel-Komponente aus dem Register Standard in diesen Streifen und stellen Sie bei `ALIGN alBottom` ein.

Selektieren Sie nun sämtliche Schaltflächen gleichzeitig, schneiden Sie sie mit CTRL X aus, aktivieren Sie mit einem Klick die Panel-Komponente und fügen Sie die Schaltflächen mit CTRL V auf dem Panel ein. Das Panel dient dazu, die Fläche der Memo-Komponente zu begrenzen. Wenn Sie jetzt nämlich die Memo-Komponente aktivieren und die Eigenschaft *ALIGN* in *aClient* ändern, füllt die Memo-Komponente immer den verbleibenden Raum bis zum Panel.

Um ein Menü einzufügen, setzen Sie die Komponente *MAINMENU* aus dem Register *STANDARD* an einem beliebigen Ort auf dem Formular ab. Ein Doppelklick auf das Icon öffnet den Menü-Editor und generiert die erste leere Kolonne des Menüs.

Tragen Sie in *Caption* *&Datei* ein. Sobald Sie diesen Eintrag mit RETURN abschliessen, wird das erste Menü der zweiten Kolonne generiert. Hier tragen Sie *&Bearbeiten* ein. Untermenüs erzeugen Sie, indem Sie das Feld *&Datei* anklicken und in das darunter entstehende Feld *&Laden* eintragen. Unter *SHORTCUT* holen Sie sich *Ctrl+O*, damit sich der Befehl auch mit diesem Tastaturkürzel aufrufen lässt. Ins darunterliegende Feld tragen Sie *&Speichern* und das Tastaturkürzel *Ctrl+S* ein. Bevor Sie auch noch *&Ende* und *Alt+F4* (von Hand, da nicht in Liste) eintragen, möchten Sie einen trennenden Strich. In dieses Feld tragen Sie deshalb einen Bindestrich ein.

In diesem Stadium haben Sie zwar bereits ein echtes Menü, das sich auf- und zuklappen lässt, aber hinter den einzelnen Menü-Items steht noch keine Funktionalität. Für das Laden und Speichern haben wir bereits Routinen geschrieben, die Sie nun wiederverwenden können. Klicken Sie auf das Menü-Item *&Laden* und wechseln Sie im Objektinspektor in das Register *EVENTS*. Statt mit einem Doppelklick eine neue Ereignisbehandlungsroutine für das einzige Ereignis *ONCLICK* zu erzeugen, holen Sie aus der Liste *btnOpenClick*. Auf dieselbe Weise verbinden Sie *&Speichern* mit der Routine *btnSaveClick*. Nur für *&Ende* haben wir noch keine Routine, weil die entsprechende Schaltfläche diese Funktionalität bereits eingebaut hat. Selektieren Sie deshalb das Menü-Item *&Ende*, ändern Sie den Namen in *mnuitEnde* und doppelklicken Sie dann in das Feld neben *ONCLICK* und tragen Sie die Anweisung *close;* ein. Schliessen Sie den Menü-Editor, speichern Sie Ihr Projekt und lassen Sie es laufen.

## 8.5 Forward- und External-Deklarationen

Prozeduren und Funktionen müssen nicht immer an jenem Ort deklariert werden, wo der Routinenkopf steht. Stattdessen kann es sich auch um eine Forward- oder eine External-Deklaration handeln.

### 8.5.1 Forward-Deklaration

Mit der Standard-Anweisung `forward` können Sie eine Prozedur oder Funktion deklarieren, wobei der Anweisungsblock erst zu einem späteren Zeitpunkt definiert werden muss.

Zwischen der Forward-Deklaration und der Definition der Prozedur ist die Deklaration anderer Prozeduren und Funktionen möglich. Auf diese Weise ist eine Rekursion (der sich überkreuzende Bezug zwischen mehreren Prozeduren) realisierbar.

Nach der Forward-Deklaration muss die Prozedur oder Funktion durch eine Deklaration des Anweisungsblocks der Routine definiert werden. Diese Deklaration kann die Parameterliste des Prozedur- oder Funktionskopfs entbehren.

### 8.5.2 External-Deklaration

Über die Standard-Anweisung `external` ist Ihr Programm in der Lage, mit gesondert kompilierten Prozeduren und Funktionen zu arbeiten, die in Assembler geschrieben wurden oder sich in DLLs befinden.


In Prozeduren und Funktionen, die aus DLLs importiert wurden, nimmt die External-Anweisung den Platz einer Deklaration sowie der Anweisungsteile ein, die ansonsten vorhanden wären. Der externe Code wird mit der Pascal-Unit oder dem Programm über die Compilerdirektive `$L` gelinkt.

## 8.6 Rekapitulation

### 8.6.1 Übung8\_1

Schreiben Sie eine Prozedur, die für einen Zahlenarray `array[1..10] of integer` die folgenden drei Werte als Integer-Zahlen zurückliefert: Durchschnitt, Minimum und Maximum. Benutzen Sie für die Prozedur einen offenen Array-Parameter, damit sich auch Arrays anderer Länge übergeben lassen (siehe unter „open-array parameters“ in der Online-Hilfe).

## 9 Klassen und Instanzen

 In diesem Kapitel schliessen Sie Bekanntschaft mit der Welt der objektorientierten Programmierung (OOP):

- Sie werden in die Philosophie und in die wichtigsten Begriffe der OOP eingeführt.
- Sie lernen, was Klassen, Instanzen und Objektfelder sind.
- Sie erfahren, wie man Methoden schreibt und wozu man Konstruktoren und Destruktoren verwendet.
- Sie arbeiten sich in fortgeschrittene Konzepte der OOP wie Vererbung sowie virtuelle und dynamische Methoden ein.




### 9.1 Objektorientierte Programmierung

**Objektorientierte Programmierung** oder OOP ist die bevorzugte Methodik, wenn es um die Entwicklung komplexer Windows-Applikationen geht. Obwohl Object Pascal objektorientierte Programmierung bestens unterstützt, ist es nicht eine rein objektorientierte, sondern eine hybride Sprache, in der sich auch in der konventionellen, prozeduralen Art entwickeln lässt.

Die drei Grundprinzipien der objektorientierten Programmierung sind:

1. Kapselung
2. Vererbung
3. Polymorphie

Diese drei Grundprinzipien lassen sich allerdings nur erklären, wenn man einige weitere Elemente und Charakteristiken der OOP einführt (genauere Begriffsdefinitionen finden Sie im nächsten Unterkapitel):

-  **Objekte:** Die zentralen Elemente bei der OOP sind Objekte.
-  **Kapselung:** Objekte sind eine Erweiterung der Records, denn sie kapseln nicht nur verschiedene Daten in Objektfeldern, sondern auch die Routinen zu deren Verarbeitung in einer Einheit.
-  **Methoden:** Die Prozeduren und Funktionen, die in Objekten gekapselt sind, nennt man Methoden.

- ☞ **Klassen und Instanzen:** Bei Objekten unterscheidet man zwischen ihrem Objekttyp, den man Klasse nennt, und einem konkreten Exemplar dieser Klasse, das man Instanz nennt. Jede Instanz eines Klassentyps besitzt seine eigene Kopie der im Klassentyp deklarierten Felder, alle Objekte benutzen jedoch dieselben Methoden gemeinsam.
- ☞ **Vererbung:** Objekte sind in eine Vererbungshierarchie eingebunden. Nachfahren übernehmen Daten und Programmcode der Vorfahrobjekte, können diese aber erweitern oder überschreiben.
- ☞ **Statische Methoden:** Statische Methoden arbeiten auf die gleiche Weise wie gewöhnliche Prozedur- oder Funktionsaufrufe. Defaultmässig sind Methoden statisch, d.h. fest mit einem Objekttyp einer bestimmten Hierarchiestufe verbunden. Der Objekttyp, dessen Methode aufgerufen werden soll, kann bereits während der Compilierung ermittelt werden.
- ☞ **Polymorphie:** Beim Prinzip der Polymorphie (Vielgestaltigkeit) erhalten Methoden mit ähnlicher Funktionalität auf verschiedenen Ebenen der Objekthierarchie den gleichen Namen, obwohl sie im Detail verschieden implementiert sind. Weil es sich um virtuelle Methoden handelt, wird erst beim Methodenaufruf zur Laufzeit der Objekttyp der übergebenen Instanz und die zugehörige Methode ermittelt. Dies spielt vor allem dort eine Rolle, wo die Methode einer Instanz aus der Methode eines Vorfahren heraus aufgerufen wird.
- ☞ **Virtuelle Methoden und spätes Binden (late binding):** Bei virtuellen Methoden steht nach der Deklaration der Befehl `virtual`. Eine Referenz auf eine virtuelle Methode wird nicht schon während der Compilierung, sondern erst während der Laufzeit in Abhängigkeit vom Objekttyp aufgelöst.  
**Tabelle virtueller Methoden** (virtual method table VMT): Das späte Binden ist nur möglich, weil für jede Klasse eine Tabelle virtueller Methoden aufgebaut wird, welche die Adressen aller virtuellen Methoden dieser Klasse verwaltet.  
**Dynamische Methoden:** Bei dynamischen Methoden steht nach der Deklaration der Befehl `dynamic`. Dynamische Methoden verwendet man genau gleich wie virtuelle. Sie unterscheiden sich von diesen nur durch den Zuteilungsmechanismus, der dazu führt, dass dynamische Methoden weniger Arbeitsspeicher als virtuelle benötigen, ihre Zuteilung dagegen etwas langsamer ist.

Viele dieser Konzepte werden erst im Laufe des Kapitels und anhand konkreter Beispiele klarer werden.

## 9.2 Begriffsdefinitionen

Im folgenden werden einige Grundbegriffe der OOP noch vertieft erklärt.

### 9.2.1 Klassen, Felder und Methoden

- ☞ Der Begriff **Klasse** wird in diesem Skript wie in den Delphi-Handbüchern synonym zu **Objektyp** verwendet. Eine Klasse ist ein Typ, der Daten und Programmcode zu einer Einheit verbindet. Die Daten, Variablen etc., werden bei der Klasse **Felder** genannt, und die zur Klasse gehörigen Prozeduren und Funktionen nennt man **Methoden**. Felder können nicht nur einfache und strukturierte Typen sein, sondern auch Instanzen von Objekten.

Klassen haben viel mit Backrezepten gemeinsam. Beide verbinden Daten und Methoden zu einer Einheit. Bei den Kochrezepten sind die Daten die Zutaten (3 Eier, 100 g Mehl) und die Methoden die Anweisungen, was damit zu tun ist („Schlagen Sie die Eier ins Mehl und rühren Sie kräftig durch“).

### 9.2.2 Instanzen

- ☞ Auch bei der Unterscheidung von Klassen und **Instanzen** gehen die Parallelen zum Backen weiter. Klassen sind nicht Dinge an sich, sondern Typen, also Rezepte für die Erstellung von Dingen. Das konkrete Exemplar einer Klasse, das zur Laufzeit existiert, nennt man dagegen Instanz. Um es zu konkretisieren: Unser Backrezept ist eine Klasse, der Kuchen, den wir damit backen, ist dagegen eine Instanz. Bevor wir eine Klasse in einem Delphi-Programm verwenden können, müssen wir erst eine Instanz, das heisst eine Variable dieses Objektyps, deklarieren und generieren.

### 9.2.3 Objekte

- ☞ Den Begriff **Objekt** haben wir bis jetzt umgangen, obwohl es um **objektorientierte Programmierung**, also um OOP geht. Der Grund ist, dass der Begriff Objekt im Gegensatz zu Klasse und Instanz nicht sehr präzise definiert ist. Das Benutzerhandbuch von Borland nennt Objekte Datentypen,

setzt sie also mit Klassen gleich. In Turbo Pascal gab es nämlich den reservierten Begriff `class` noch nicht, sondern damals hiess es `object`.

Gewisse andere Autoren brauchen den Begriff dagegen synonym zu Instanz. Häufig ist die Bezeichnung Objekt dort anzutreffen, wo es keine Rolle spielt, ob es sich um eine Klasse oder die dazugehörige Instanz handelt. (Der Begriff „Schwarzwäldertorte“ kann ja auch sowohl für diesen Typ Torte wie für ein ganz konkretes Exemplar davon verwendet werden.) Wir schliessen uns diesem Sprachgebrauch an: Im Skript wird Objekt dann verwendet, wenn es sowohl um die Klasse wie um die Instanz geht, während die präziseren Begriffe benutzt werden wenn es nur um die Klasse oder nur um die Instanz geht.

#### 9.2.4 Eigenschaften

☞ **Eigenschaften** (properties) sind ein weiteres zentrales Element von Objekten, mit dem Sie bereits mehrmals gearbeitet haben. Beispiele von Eigenschaften sind die Höhe eines Formulars, die Beschriftung eines Buttons oder die Grösse einer Schriftart usw.

Eigenschaften sehen zwar wie Objektfelder aus, sind aber eine natürliche Erweiterung von diesen. Sowohl Objektfelder wie Eigenschaften können dazu benutzt werden, Attribute eines Objekts auszudrücken. Aber während Felder Elemente sind, die sich direkt untersuchen und ändern lassen, verschaffen Eigenschaften mehr Kontrolle über den Zugriff auf Attribute.

Hinter Eigenschaften stehen ein privates Objektfeld, in dem der Attributwert gespeichert wird, und zwei spezielle Zugriffsmethoden, mit denen lesender- oder schreibenderweise auf das private Objektfeld zugegriffen werden kann. In diesen Methoden können auch weitere Seiteneffekte implementiert sein. Eine Zuweisung an eine Eigenschaft löst somit einen impliziten Aufruf der für das Schreiben zuständigen Methode aus.

#### 9.2.5 Komponenten und OOP

Sie kennen diese OOP-Elemente von den Übungen her bereits alle: Die Komponenten in Delphi sind Objekttypen. Wenn Sie ein Formular erzeugen, dann generieren Sie eine Instanz eines Objekts, das von der Klasse `TForm` abstammt. Dank den Methoden, die das Formular von seinem

Vorfahr erbt, können Sie es anzeigen, in der Grösse verändern oder schliessen, ohne eine einzige Zeile Programmcode zu schreiben.

Die Komponenten, mit denen Sie das Formular erweitern, sind Objektfelder. Die Ereignisbehandlungsroutinen, die Sie bis jetzt geschrieben haben, sind Methoden, die zum Objekttyp Ihres Formulars oder Ihrer Komponente gehören.

Wenn Sie während der Entwicklung Formulare und andere Komponenten gestalten, indem Sie im Register `PROPERTIES` des Objektinspektors neue Werte eintragen, dann ändern Sie veröffentlichte Eigenschaften ihres Formulars.

Sie sehen, dass die objektorientierte Programmierung das rasche Entwickeln neuer, stabiler Applikationen aus vorgefertigten Komponenten erheblich vereinfacht, ja vielleicht überhaupt erst möglich gemacht hat.

### 9.3 Die Syntax einer Klasse

Damit Sie nicht nur auf vorhandene Klassen zugreifen, sondern auch Ihre eigenen Klassen deklarieren können, müssen Sie mit der Syntax einer Klasse vertraut sein.

Eine Klasse oder einen Objekttyp deklarieren Sie global. Eine Klasse kann nicht im Deklarationsteil einer Variablen oder innerhalb einer Prozedur, Funktion oder eines Methodenblocks deklariert werden. Die Deklaration einer Klasse steht deshalb im Interface-Teil einer Unit innerhalb eines mit `type` eingeleiteten Typendeklarationsteils. Das reservierte Wort `class` nach Typenbezeichner und Gleichzeichen macht ersichtlich, dass es sich hier um den Kopf einer Klassendeklaration handelt. Dahinter steht in Klammern der Typ des Vorfahren. Klassen bei denen diese Klammer fehlt, stammen automatisch von der Urmutter aller Klassen, `TObject`, ab.

Nach diesem Klassenkopf folgt eine meist lange, eingerückte Auflistung der dazugehörenden Objektfelder und Methoden. Bei den Objektfeldern finden sich die folgenden vier Elemente: Feldbezeichner, Doppelpunkt, Typ des Feldes (meist auch eine Klasse) und Strichpunkt. Für die Methoden wird nur der mit einem Strichpunkt abgeschlossene Methodenkopf, also `procedure` oder `function`, Methodenbezeichner und Parameterliste, aufgenommen. Die Implementierung der Methode erfolgt im Implementationsteil der Unit.



Die letzte Zutat einer Klassendeklaration sind die Standardanweisungen `private`, `protected`, `public` und `published`, welche die Deklaration in vier Teile aufspalten. Die Elemente, die in diesen vier Teilen aufgeführt sind, zeichnen sich durch unterschiedliche **Sichtbarkeit** aus. Die Sichtbarkeit entscheidet darüber, ob man auf diese Elemente von einer anderen Unit, von jeder beliebigen Klasse oder nur von einem Nachkommen aus zugreifen kann. Kapitel 11 wird sich diesem Thema noch ausführlich widmen.

Das folgende Codebeispiel zeigt die etwas gekürzte Klassendeklaration für `TButton` (in der Professional- und der Client/Server-Ausgabe von Delphi 2.0 ist der Quellcode für die VCL dabei):

```

1  type
2  TButton = class(TButtonControl)
3      private
4          FDefault: Boolean;
5          FCancel: Boolean;
6          FActive: Boolean;
7          FReserved: Byte;
8          FModalResult: TModalResult;
9          procedure SetDefault(Value: Boolean);
10         ...
11     protected
12         procedure CreateParams(var Params:
13             TCreateParams); override;
14         procedure CreateWnd; override;
15         procedure SetButtonStyle(ADefault:
16             Boolean); virtual;
17     public
18         constructor Create(AOwner: TComponent);
19             override;
20         procedure Click; override;
21     published
22         property Cancel: Boolean read FCancel
23             write FCancel default False;
24         property Caption;
25         ...
26 end;
```

Das reservierte Wort `type` in Zeile 1 muss nicht vor jeder Klassendeklaration stehen, sondern nur vor der ersten Deklaration in einem Typendeklarationsteil.

In Zeile 2 erfahren wir, dass hier die Klasse `TButton` als Nachfahr von `TButtonControl` deklariert wird. Die Sichtbarkeitsattribute in Zeile 3, 11, 17 und 21 teilen die Deklaration in vier Teile auf. Die Zeilen 4 bis 8 deklarieren `private`

Felder. In Zeile 18 wird eine ganz spezielle Methode deklariert, nämlich der **Konstruktor**. Konstruktoren und ihre Gegenstücke, die **Destruktoren**, stehen immer im Public-Teil. Im nächsten Unterkapitel über Methoden erfahren Sie mehr darüber. Im Teil `published` ab Zeile 21 finden Sie dann noch zwei der unzähligen Eigenschaften, die diese Komponente aufweist.



Wechseln Sie in Ihr Editor-Projekt und suchen Sie im Quelltexteditor im oberen Teil der Unit `uMainEdi` die Deklaration der Klasse `TfrmEditor`. Analysieren Sie die einzelnen Felder und Methoden dieser Klasse.

Bis jetzt haben Sie erst das Backrezept, nämlich die Deklaration des Klassentyps. Bevor Sie damit etwas anfangen können, müssen Sie eine Variable dieses Typs deklarieren, z.B. in der folgenden Art.

```
var  
    MeinKnopf : TButton;
```

Im Gegensatz zu Records und anderen statischen Typen haben Sie damit aber noch keinen Speicherplatz für Ihre Instanz reserviert. Dies geschieht erst mit dem Aufruf des Konstruktors. Diese spezielle Methode werden Sie im nächsten Unterkapitel kennenlernen.

## 9.4 Methoden

Klassen sind eine Art erweiterte Recordtypen. Wie diese fassen sie verschiedenartige Daten zu einer Einheit zusammen. Zu Klassen gehören im Unterschied zu Records aber auch Methoden, das heisst Prozeduren und Funktionen, die nur für die Verarbeitung der Daten der zugehörigen Klasse zuständig sind.

### 9.4.1 Methoden allgemein

Wie Sie bereits vernommen haben, gehört zu einer Methode erstens eine Methodendeklaration innerhalb der Deklaration der zugehörigen Klasse im Interface-Teil einer Unit und zweitens eine Implementierung im Implementationsteil derselben Unit.

Die Deklaration einer Methode entspricht etwa der Deklaration einer Prozedur oder Funktion. Allerdings gibt es die folgenden Unterschiede:

- Für Methoden ist die Deklaration im Gegensatz zu Prozeduren und Funktionen nicht optional, sondern zwingend.
- Die Deklaration einer Methode erfolgt innerhalb der Deklaration der zugehörigen Klasse.
- Sie kann mit einer der drei Standardanweisungen `virtual`, `dynamic` oder `override` versehen sein.

Mit diesen drei Standardanweisungen dringen wir in fortgeschrittene Konzepte von OOP vor. Sie müssen deshalb noch bis zum Kapitel 9.7 warten.

Die Implementierung der Methode erfolgt im Implementationsteil der Unit und ist von zwei Unterschieden abgesehen identisch mit der Implementierung einer normalen Funktion oder Prozedur:

1. Der erste Unterschied besteht darin, dass der Methodenkopf im Implementationsteil einen **qualifizierten Bezeichner**, d.h. dem Namen der zugehörigen Klasse, gefolgt von einem Punkt, enthält. Der qualifizierte Bezeichner in Zeile 3 des folgenden Beispiels ist fett markiert.

```

1  implementation
2  {$R *.DFM}
3  procedure TfrmEditor.btnOpenClick
4      (Sender: TObject);
5      ...

```

2. Der zweite Unterschied besteht darin, dass bei jedem Methodenaufruf das zugehörige Objekt implizit übergeben wird. Die Methode kann also auf alle Datenfelder eines Objekts zugreifen und Veränderungen an ihnen vornehmen, ohne dass man die Datenfelder in der Parameterliste übergeben muss.

Bis jetzt haben Sie in den Übungen vor allem eine Art von Methoden kennengelernt, nämlich **Ereignisbehandlungsroutinen**. Ereignisbehandlungsroutinen sind spezialisierte Methoden, die automatisch aufgerufen werden, wenn das zugehörige Ereignis eintritt. **Ereignisse** repräsentieren Aktionen des Benutzers bzw. der Benutzerin oder interne Systemnachrichten, die Ihr Programm erkennen kann, beispielsweise ein Mausklick. Für Ereignisbehandlungsroutinen gibt es also keinen expliziten Methodenaufruf.

Es kann aber trotzdem vorkommen, dass Sie Methoden schreiben möchten, die nicht automatisch von Ereignissen ausgelöst, sondern konventionell aufgerufen werden. Dies ist mit einem **qualifizierten Methodenaufruf** möglich. Wie

sonstige Prozedur- und Funktionsaufrufe besteht ein Methodenaufruf aus dem Namen der Methode, gefolgt von der aktuellen Parameterliste. Der einzige Unterschied zu einem konventionellen Prozedur- oder Funktionsaufruf besteht darin, dass die Methode natürlich nur von der Instanz ihrer Klasse oder eventuell von deren Nachfahren aufgerufen werden darf und deshalb durch Voranstellen des Namens der Instanz als zu dieser Instanz zugehörig gekennzeichnet wird. Hier der Aufruf der Methode `Draw` als Beispiel:

```
frmHauptformular.Canvas.Draw(0, 0, Bild1);
```

#### 9.4.2 Konstruktoren und Destruktoren

☞ **Konstruktoren** und **Destruktoren** sind spezielle Methoden zum Erzeugen und Freigeben von Objekten. Objekte sind nämlich wie Zeiger dynamische Sprachelemente, die nicht schon zu Beginn der Laufzeit eine bestimmte Menge Speicherplatz fest reservieren, sondern den Speicher je nach Bedarf beanspruchen und wieder freigeben. Dazu dienen die Konstruktoren und Destruktoren.

Eine Klasse kann keinen, einen oder mehrere Konstruktoren und Destruktoren für die Objekte des Klassentyps haben. Die reservierten Wörter `constructor` und `destructor` stehen am Anfang jeder Deklaration an Stelle von `procedure` oder `function`. Konstruktoren und Destruktoren lassen sich vererben wie andere Methoden auch.

##### **Konstruktoren**

Konstruktoren werden benutzt, um neue Objekte zu erzeugen und zu initialisieren. Dazu muss der Konstruktor über den Bezeichner der Klasse (statt der Instanz) aktiviert werden. Wenn ein Konstruktor über den Bezeichner der Klasse aktiviert wird, geschieht folgendes:

- Speicherplatz für das neue Objekt wird auf dem Heap zugewiesen.
- Der zugewiesene Speicherbereich wird initialisiert. Alle ordinalen Werte werden auf Null, alle Zeiger und Objektinstanzen werden auf nil und alle String-Felder werden auf leer gesetzt.
- Die benutzerdefinierten Aktionen des Konstruktors werden ausgeführt.

- Eine Referenz auf den neu zugewiesenen und initialisierten Speicherbereich wird vom Konstruktor zurückgegeben.

Folgendermassen können Sie etwa den vorhin deklarierten Button erzeugen:

```
var
  MeinKnopf: TButton;
begin
  MeinKnopf := TButton.Create(frmHaupt);
end;
```

Falls für eine Klasse bereits eine Instanz existiert, lässt sich der Konstruktor auch über eine Objektreferenz, d.h. den Namen der Instanz, aufrufen. In diesem Fall werden nur die benutzerdefinierten Anweisungen des Konstruktors ausgeführt, aber es wird kein neues Objekt erzeugt.

Der erste Befehl eines Konstruktors ist häufig der Aufruf eines ererbten Konstruktors, um die ererbten Felder des Objektes zu initialisieren. Danach initialisiert der Konstruktor die neu eingeführten Felder der Klasse.

### ***Destruktoren***

Destruktoren werden verwendet, um Objekte aus dem Speicher zu entfernen. Wenn ein Destruktor aufgerufen wird, so werden die benutzerdefinierten Aktionen ausgeführt, dann wird der dem Objekt zugewiesene Speicherbereich freigegeben. Die benutzerdefinierten Aktionen eines Destruktors sind normalerweise die Löschung von eingebetteten Objekten und das Freigeben von Ressourcen, die von dem Objekt belegt wurden.

Die letzte Aktion eines Konstruktors ist typischerweise der Aufruf des ererbten Destruktors, um alle ererbten Objektfelder zu löschen.

#### **9.4.3 Eine eigene Suchmethode erstellen**

Um die Theorie etwas aufzulockern, werden Sie nun eine etwas längere Ereignisbehandlungsroutine schreiben.



Wechseln Sie in Ihr Editor-Projekt. Fügen Sie in das Formular eine Komponente `FINDDIALOG` aus dem Register `DIALOGS` ein und nennen Sie diese Komponente *dlgSuchen*. Öffnen Sie mit einem Klick auf das Pluszeichen die Eigenschaft `OPTIONS` und setzen Sie `FRHIDEMATCHCASE`, `FRHIDEWHOLEWORD` und `FRHIDEUPDOWN` auf *true*. Damit erhalten Sie einen Suchdialog

ohne die Optionen „Gross-/Kleinschreibung“, „Ganzes Wort“ und „Suche aufwärts/abwärts“.


Öffnen Sie den Menü-Editor mit einem Doppelklick auf die Menü-Komponente. Fügen Sie in der Kolonne &Bearbeiten das Menü-Item *&Suchen* an, geben Sie im den Namen *mnuitSuchen* und weisen Sie ihm das Tastaturkürzel *Ctrl+F* zu. Erstellen Sie mit Doppelklick auf ONCLICK eine Prozedur *TfrmEditor.mnuitSuchenClick* für das Menü-Item und tragen Sie folgende Anweisung ein:

*dlgSuchen.Execute;*

Dieser Befehl öffnet nur ein Dialogfenster und nimmt den Suchstring entgegen, führt aber keine Suche durch. Die eigentliche Suchmethode plazieren Sie in der Komponente *dlgSuchen* als Ereignisbehandlungsroutine von ONFIND. Ergänzen Sie diese Routine, bis sie folgendermassen aussieht:

```

1  procedure TfrmEditor.dlgSuchenFind
2      (Sender: TObject);
3  var
4      strSuchText  : string;
5      strQuelle     : string;
6      carLaenge    : cardinal;
7      carPosition  : cardinal;
8  begin
9      strSuchText := dlgSuchen.FindText;
10     carLaenge   := Length(strSuchText);
11     strQuelle    := frmEditor.memEditor.Text;
12
13     carPosition := Pos(strSuchText, strQuelle);
14     If carPosition > 0 then begin
15         frmEditor.BringToFront;
16         frmEditor.ActiveControl := memEditor;
17         memEditor.SelStart := carPosition - 1;
18         memEditor.SelLength := carLaenge;
19     end else begin
20         MessageDlg(
21             '"' + strSuchText + '" nicht gefunden',
22             mtInformation, [mbOK], 0);
23     end;
24 end;
```

 In der ersten Zeile implementieren Sie den Methodenkopf. Beachten Sie, dass vor dem Methodennamen der Name der zugehörigen Klasse und ein Punkt stehen. In der Parameterliste steht nur ein einziger Wertparameter, nämlich **Sender** vom Typ *TObject*. Dieser Parameter informiert darüber, welche Komponente das Ereignis empfangen und deshalb die Ereignisbehandlungsroutine aufgerufen hat.

In Zeile 4 bis 7 deklarieren Sie vier lokale Variablen. Im Anweisungsteil holen Sie in Zeile 9 erst einmal den Suchtext von der Komponenten `dlgSuchen`. Mit der Funktion `Pos` versuchen Sie in Zeile 13, den gesuchten String im Text der Memokomponente zu lokalisieren. Falls dies gelingt (`carPosition > 0`), dann holen Sie das Hauptformular in den Vordergrund, aktivieren die Memokomponente und markieren in Zeile 17 und 18 den gefundenen Text. Andernfalls gibt die Methode in Zeile 20 bis 22 eine Mitteilung aus, dass der Suchtext nicht gefunden werden konnte.

## 9.5 Die Klasse `TApplication`

Mit Ihrem neu erworbenen Wissen über Klassen und Methoden können Sie sich jetzt die Anweisungen in der Projektdatei, z.B. jene des Editorprojekts, noch einmal ansehen. Sie erinnern sich: Diese Datei enthielt ausser Deklarationen nur die folgenden drei Anweisungen:

```
1 Application.Initialize;  
2 Application.CreateForm(  
3   TfrmEditor, frmEditor);  
4 Application.Run;
```

Jedes Delphi-Programm benutzt automatisch eine Komponente `TApplication`, die Ihr Programm einkapselt. Delphi deklariert eine Variable `Application` vom Typ `TApplication`, die eine Instanz Ihres Programms ist.

Die Methode `Initialize` ist das erste, was für jedes Delphi-Projekt aufgerufen wird. Abgesehen von Anwendungen, die einen OLE-Automatisierungs-Server enthalten, führt `Initialize` keinerlei Aktion durch. Bei Projekten, die keine OLE-Automatisierungs-Server enthalten, können Sie den Aufruf von `Initialize` ohne Probleme löschen.

In den nächsten Zeilen erzeugt das Hauptprogramm die zur Anwendung gehörigen Formulare. In unserem Fall ist dies bis jetzt nur ein einziges Formular, nämlich `frmEditor` in Zeile 2 und 3. Die Methode `CreateForm` erzeugt ein neues Formular des Typs, der durch den ersten Parameter angegeben wird und weist es der Variablen, die durch den zweiten Parameter gegeben ist, zu. Der Besitzer des neuen Formulars ist die Instanz `Application`. Das Formular, das durch den ersten Aufruf von `CreateForm` in einem Projekt erzeugt wird, wird zum Hauptformular des Projekts. Wenn Sie Ihr Programm ausführen, wird die Programmethode `Run` aufgerufen.

## 9.6 Vererbung

- ☞ Ein Objekttyp kann Komponenten von einem anderen Klassentyp erben. Die erbende Klasse ist ein **Nachkomme**, die vererbende Klasse ist ein **Vorfahr**.

Vererbung ist transitiv, wenn z.B. T3 von T2 erbt und T2 von T1 erbt, dann erbt T3 auch von T1. Der Bereich eines Klassentyps erstreckt sich über den Klassentyp selbst und alle seine Nachkommen.

Eine Nachkommenklasse enthält implizit alle Komponenten, die von ihren Vorfahrenklassen definiert worden sind. Eine Nachkommenklasse kann zu den geerbten Komponenten neue hinzufügen, aber sie kann die Definition einer Komponente in einer Vorfahrenklasse nicht rückgängig machen.

Der vordefinierte Klassentyp `TObject` ist der oberste Vorfahr aller Klassentypen. Wenn die Deklaration eines Klassentyps keinen Vorfahrentypen spezifiziert (d.h. der Vererbungsteil in der Deklaration der Klasse wurde ausgelassen), wird der Klassentyp von `TObject` abgeleitet. `TObject` ist in der Unit `System` deklariert und definiert eine Anzahl von Methoden, die auf alle Klassen angewendet werden.

## 9.7 Virtuelle und dynamische Methoden

### 9.7.1 Polymorphie

- ☯ In diesem Schritt geht es nun um den dritten und faszinierendsten Pfeiler der OOP, um die **Polymorphie** (Vielgestaltigkeit). Dahinter verbirgt sich die Idee, virtuelle Methoden mit gleichem Bezeichner auf alle Instanzen einer vererbten Objekthierarchie anwenden zu können, obwohl sie innerhalb dieser Methoden eben unterschiedlich reagieren.

Dies abstrakten Gesetze treffen wir auch in der Wirklichkeit an, wo z.B. jedes Fortbewegungsmittel die Methode `goAway` kennt, jedoch anders darauf reagiert. Eben diese Fähigkeit, gleichartige Prozesse in unserer Umwelt zu erkennen und aus ihnen ein abstraktes Prinzip herzuleiten, ist die Motivation der Polymorphie. Die Vielgestaltigkeit soll also dem Entwickler helfen, die Informatik näher an die Realität zu bringen.



### 9.7.2 Virtuelle Methoden und late binding

Diese zur Laufzeit generierte Flexibilität ist aber nur möglich, wenn die Referenz auf eine Methode nicht schon beim Kompilieren, sondern erst während der Laufzeit in Abhängigkeit des Typen festgelegt wird. Object Pascal (eigentlich auch alle anderen OOP-Sprachen) löst das Problem des **late binding** mit Hilfe der sogenannten Virtual Method Tables (VMT), die ein Teil jeder Klasse ist. Das sind Tabellen mit Methodenzeigern, die für jede Klasse mit Hilfe des Konstruktors automatisch angelegt werden, sofern die Klasse über virtuelle Methoden verfügt. Nun genug der Abstraktion, anhand des folgenden konkreten Beispiels steuern wir auf den Kern der Sache zu, denn Programmieren geht über Studieren.

Stellen Sie sich eine abstrakte Oberklasse Fortbewegungsmittel vor, die sich in Zeile 18 befindet. Von dieser Klasse vererben wir je drei neue Fortbewegungsmittel, nämlich ein Flugzeug (Zeile 24), ein Auto (Zeile 29) und ein Velo (Zeile 34). Selbstverständlich kommt nun eine virtuelle Methode namens `goAway` dazu, welche in der Basisklasse `virtual` und in den abgeleiteten `override` deklariert wird und unterschiedlich reagiert. Und genau das ist der Punkt:

Beim Aufruf der einzelnen Methoden `ShowMove` in Zeile 87 bis 89 entscheidet nun die Methode `ShowMove` dynamisch zur Laufzeit in Zeile 46, wie darauf zu reagieren ist. Das heisst ich kann ständig Instanzen mit dem gleichen Bezeichner aufrufen, die sich dann vielgestaltig manifestieren.

```

1  unit Polymrph;
2
3  interface
4
5  uses
6      SysUtils, WinTypes, WinProcs, Messages,
7      Classes, Graphics, Controls, Forms, Dialogs,
8      StdCtrls;
9
10 type
11     TForm1 = class(TForm)
12         Button1: TButton;
13         Edit1: TEdit;
14         procedure Button1Click(Sender: TObject);
15     end;
16
17     TVehicle = class(TObject)

```

```
18     procedure showMove;
19     procedure goAway; virtual;
20 end;
21
22 TPlane = class(TVehicle)
23     procedure goAway; override;
24 end;
25
26 TCar = class(TVehicle)
27     procedure goAway; override;
28 end;
29
30 TBicycle = class(TVehicle)
31     procedure goAway; override;
32 end;
33
34 var
35     Form1: TForm1;
36
37 implementation
38 {$R *.DFM}
39
40 procedure TVehicle.showMove;
41 begin;
42     goAway;           {Polymorphie!}
43 end;
44
45 procedure TVehicle.goaway;
46 begin
47     messageDlg('Fly something
48     else',mtInformation,[mbOk],0);
49 end;
50
51 procedure TPlane.goaway;
52 begin;
53     MessageDlg('Fly the digital freedom.
54     ',mtInformation,[mbOk],0);
55 end;
56
57 procedure TCar.goaway;
58 begin;
59     MessageDlg('Drive the crash car.
60     ',mtInformation,[mbOk],0);
61 end;
62
63 procedure TBicycle.goaway;
64 begin;
65     MessageDlg('Ride on a Klein Bike.
66     ',mtInformation,[mbOk],0);
67 end;
```


```
68
69 procedure TForm1.Button1Click(Sender:
70 TObject);
71 var
72     Plane: TPlane;
73     Car: TCar;
74     Bicycle: TBicycle;
75     AvailableDays: Integer;
76 begin
77     car:=TCar.create;
78     plane:=Tplane.create;
79     bicycle:= TBicycle.create;
80
81     AvailableDays := StrToInt(Edit1.Text);
82     case AvailableDays of
83         1..3 : Plane.showMove;
84         4..6 : Car.showMove;
85         else Bicycle.showMove;
86     end;
87 end;
88 end.
```

## 9.8 Rekapitulation

### 9.8.1 Kapiteltest

- 1) Erläutern Sie Unterschiede und Gemeinsamkeiten von Records und Klassen.
- 2) Richtig oder falsch?
  - a) Der Aufruf eines Konstruktors initialisiert automatisch alle Elemente einer Instanz.
  - b) Dynamische Methoden ruft man gleich auf wie virtuelle, sie haben aber andere Auswirkungen auf die Performance der Applikation.
  - c) Jede Instanz einer Klasse besitzt eine eigene Kopie der im Klassentyp deklarierten Felder und Methoden.
  - d) Eine Nachkommenklasse enthält alle Felder ihres Vorfahren. Sie kann neue hinzufügen oder geerbte Felder entfernen.
- 3) Gibt es Polymorphie ohne Vererbung?


## 10 Eigenschaften und Windows-Botschaften

 Nun wenden wir uns der komplexeren Materie, sprich der Steuerung im Hintergrund sowohl von Delphi wie auch von Windows zu:

- Sie lernen, mit den automatischen Mechanismen der Eigenschaften umzugehen,
- Sie verstehen die Technik der Windows-Botschaften und
- Sie lernen das Definieren von eigenen Botschaften.

### 10.1 Was sind Eigenschaften (Properties)?

Wenn wir jetzt künftig von **Properties** sprechen, sind eigentlich auch Eigenschaften gemeint. Denn die Properties sind eine der besonders wichtigen sprachlichen Neuerungen von Delphi und verkörpern sozusagen das **OOP-Paradigma** welches besagt:

 Ein Property einer Instanz sollte man nur über die dazugehörige Klassenmethode verändern können.

Und genau dazu dienen die Properties. Denn die Daten sind in den Properties mit definierten Methoden verbunden, die diese Daten verändern können. Jedesmal, wenn Sie einem Property einen Wert zuweisen oder das Property nur lesen (z.B. den Wert einer anderen Variablen zuweisen) ruft Delphi automatisch die entsprechende Methode auf, so dass diese auf jede Änderung reagieren kann. Somit sind Properties also nur vereinfachte Methodenaufrufe.

#### 10.1.1 Vorteile von Properties

Properties vermitteln dem Anwender die Illusion, dass er den Wert einer Variablen in der Komponente setzt oder ausliest. Dem Entwickler hingegen ermöglichen sie, die zugrunde liegende Datenstruktur zu verbergen oder Nebeneffekte (Side Effects) des Zugriffs zu vermeiden.

Der Einsatz von Properties bietet dem Entwickler mehrere Vorteile:

- Properties können Werte oder Formate prüfen, während der Anwender diese zuweist. Die Validierung der Anwen-

dereingaben beugt Fehlern vor, die man durch ungültige Werte verursachen kann.

- Das Programm kann bei Bedarf geeignete Werte erzeugen.

Der vielleicht häufigste Fehlertyp ist das Referenzieren einer Variable, welche keinen Anfangswert besitzt. Indem Sie den Wert zu einem Property erklären (**default**), können Sie jedoch sicherstellen, dass der aus dem Property ausgelesene Wert immer gültig ist.

Im Gegensatz zu einem Feld kann ein Property Details seiner Implementierung vor dem Benutzer verbergen. Beispielsweise lassen sich Daten intern verschlüsseln, für das Setzen oder Lesen des Property aber unverschlüsselt anzeigen. Obwohl der Wert eines Property sich nach aussen nur als einfache Zahl darstellt, könnte die Komponente diesen Wert in einer Datenbank abfragen oder mit komplizierten Verfahren errechnen.

Properties bieten auch erhebliche Vorteile im Komponentenbau, sowohl für Sie als Komponentenentwickler als auch für die Benutzer Ihrer Komponenten. Properties lassen sich im Entwurfsmodus im Objektinspektor anzeigen (**published**), darin liegt ihr offensichtlichster Vorzug. Denn auch Ihre Programmierarbeit wird dadurch erleichtert; Sie brauchen nämlich nicht eine Menge von Parametern zu verarbeiten, um ein Objekt zu konstruieren, Sie brauchen lediglich die vom Benutzer zugewiesenen Werte einzulesen.

#### 10.1.2 Properties selbst deklarieren

Die Deklaration eines Property und ihrer Implementierung ist einfach und konsistent.

Die Deklaration eines Property enthält drei Dinge:

- Den Namen des Property
- Den Typ des Property
- Methoden für das Lesen und/oder Schreiben des Property-Wertes

Die Properties einer Komponente **sollten wenigstens im Abschnitt public deklariert sein**, was ein einfaches Lesen und Schreiben des Property von ausserhalb der Komponente zur Laufzeit ermöglicht.

So sieht eine typische Property-Deklaration aus:

```
type TMyClass = class(TPersistent)
private
  FColor: TColor; {Feld interne Speicherung}
  function GetColor: TColor; {read-Methode }
  procedure SetColor(newColor: TColor);
                                     {write-Methode}
public
  property Color: TColor read GetColor write
                                     SetColor;
```

### ***Die Methode read***

Die Methode `read` eines Property ist eine Funktion, die keine Parameter annimmt, und einen Wert vom Typ des Property zurückliefert. Gemäss Konvention ist der Name dieser Funktion `Get`, gefolgt vom Namen des Property. Die Read-Methode für ein Property mit dem Namen `Color` würde also `GetColor` heissen.

Die Regel „keine Parameter“ hat allerdings eine Ausnahme, nämlich bei Array-Properties, die ihre Indizes als Parameter übergeben (siehe Array-Properties, 10.1.4).

Wenn Sie keine Read-Methode deklarieren, erlaubt die Eigenschaft nur Schreibzugriff. Properties mit ausschliesslichem Schreibzugriff sind sehr selten und in der Regel auch nicht sinnvoll.

```
Function TMyClass.GetColor: TColor;
begin
  Result:= FColor;
end;
```

### ***Die Methode write***

Die Methode `write` eines Property ist immer eine Prozedur, die genau einen Parameter annimmt. Der Parameter muss vom gleichen Typ sein wie das Property. Er lässt sich durch `call by value` oder über `call by reference` übergeben und jeden beliebigen Namen haben. Gemäss Konvention lautet der Name der Prozedur `Set`, gefolgt vom Namen des Property. Die Write-Methode für ein Property mit dem Namen `Color` würde also `SetColor` heissen.

Der im Parameter übergebene Wert wird verwendet, um den neuen Wert des Property zu setzen. Daher muss die Write-

Methode fähig sein, alle erforderlichen Behandlungen der Werte durchzuführen, um sie in der richtigen Form in das interne Speicherfeld zu schreiben. Wird keine Write-Methode deklariert, gestattet die Eigenschaft nur Lesezugriff. Üblicherweise wird **vor dem Setzen des Wertes** überprüft, ob sich der neue Wert vom aktuellen unterscheidet.

```
procedure TMyClass.SetColor(newColor:
                               TColor);
begin
    if newColor <> FColor then begin
        FColor := newColor;
        Invalidate;
    end;
end;
```

### 10.1.3 Standardwerte von Properties

Beim Deklarieren eines Properties lässt sich optional ein Standardwert deklarieren. Der Standardwert für die Eigenschaft einer Komponente ist der Wert, der für diese Eigenschaft im Konstruktor der Komponente gesetzt ist. Wenn Sie zum Beispiel eine Komponente aus der Komponentenpalette in ein Formular einfügen, erzeugt Delphi die Komponente durch Aufrufen des Konstruktors der Komponente, der den Anfangswert des Property der Komponente festlegt.

Zum Deklarieren eines Standardwerts für ein Property fügen Sie den Befehl **default** an die Deklaration (oder Neudeklaration) des Property an, gefolgt vom Standardwert.

```
property Color: TColor read GetColor write
    SetColor; default clBlue;
```

- ☞ Die Deklaration eines Standardwerts in der Deklaration setzt den Property nicht automatisch auf diesen Wert. Sie als Entwickler müssen sicherstellen, dass der Konstruktor der Klasse das Property tatsächlich auf den eingegebenen Wert setzt.

```

Constructor TMyClass.Create(AOwner:
                                TComponent);

begin
  inherited Create(AOwner);
  FColor:= clBlue;
end;

```

#### 10.1.4 Erzeugen von Array-Properties

In der Regel haben Read-Methoden keine Parameter, hier folgt nun die mögliche Ausnahme:

Einige Properties bieten sich für eine Indizierung an, im Stil von Arrays. Sie bestehen dann aus mehreren Werten, die einer Art von Indexwert entsprechen. Ein Beispiel bei den Standardkomponenten ist das Property `Lines` der Memo-Komponente. `Lines` ist eine indizierte Liste von Strings, die den Text des Memos, siehe unser Beispiel-Editor, ausmachen, und diese Liste lässt sich wie ein String-Array behandeln. In diesem Fall gibt das Array-Property dem Benutzer Zugriff auf ein bestimmtes Element (einen String) in einer grösseren Datenmenge (dem Memotext).

Array-Properties funktionieren genau so wie andere Properties und lassen sich im wesentlichen in der gleichen Weise deklarieren. Ein Array-Property funktioniert nach aussen hin wie ein normales Array, Sie greifen also beispielsweise mit

```
color[0] := clWhite;
```

auf das nullte Element des Properties `Color` zu (eigentlich müsste es jetzt genaugenommen `Colors` heissen). Bei der Deklaration unterscheidet das fehlende `of` und das Anreihen von Dimensionen das Array-Property von einem normalen Array:

```

property Color[n: Byte]: TColor read
    GetColor write SetColor;

```

Die Schreib- und Lesemethoden sind nun jeweils um einen oder mehrere Indexparameter erweitert:

```

function GetColor(n: Byte): TColor;
procedure SetColor(n:Byte; newColor:
    TColor):

```

#### 10.1.5 Ereignisse sind auch Properties

Abschliessend noch eine Ergänzung, die bereits jetzt ihr Urvertrauen in Delphi festigen sollte:



Komponenten verwenden auch Properties, um ihre Ereignisse zu implementieren. Anders als die meisten anderen Properties verwenden Ereignisse keine Methoden, um ihre Funktionen `read` und `write` zu implementieren. Ereignis-Properties arbeiten stattdessen mit einem privaten Feld vom gleichen Typ wie die Properties.

Zum Beispiel speichert der Compiler den Zeiger der Methode `OnClick` in einem Feld namens `FOnClick` des Typs `TNotifyEvent`. Die Deklaration des Ereignis-Property `OnClick` sieht wie folgt aus:

```

1  type TControl = class(TComponent)
2  { Feld deklarieren, das den Methodenzeiger
3    enthalten soll }
4  private
5    FOnClick: TNotifyEvent;
6  protected
7    property OnClick: TNotifyEvent read
8                                     FOnClick write FOnClick;
9  end;
10
```

Wie bei jeder anderen Property können Sie auch den Wert eines Ereignisses zur Laufzeit setzen oder ändern. Der Hauptvorteil, dass Ereignisse auch Properties sind, liegt jedoch darin, **dass Anwender Ereignissen Behandlungsroutinen mit dem Objektinspektor im Entwurfsmodus direkt zuweisen können.**

## 10.2 Windows Nachrichten

### 10.2.1 Windows API Funktionen

Das **Windows-API** stellt quasi eine Art Toolbox dar (eigentlich ist es die grösste Toolbox der Welt), mit der Sie Anwendungen erstellen. Es umfasst insgesamt mehr als 700 Funktionen. Ein ganze Reihe, **sicherlich weniger als 50**, aktiviert Ihr Programm dauernd, den Rest benötigen Sie oder Delphi nur sehr selten oder gar nicht. Die verschiedenen Funktionen des APIs sind nach bestimmten Kriterien in unterschiedlichen Bibliotheken verteilt. Die Funktionen werden aber nicht beim Linken hinzugefügt und fest ins Programm eingebunden, sondern erst zur Laufzeit bei Bedarf dynamisch ins System geladen und ausgeführt. Aus diesem Grund besitzen solche Bibliotheken auch einen besonderen

☞ Namen, der mit dem Begriff **DLL** einhergeht und für Dynamic Link Library (= Dynamische Laufzeitbibliothek) steht.

DLLs für Windows besitzen üblicherweise die Dateinamenerweiterungen EXE, DLL, oder DRV. Zur ersten Gruppe gehören die Bibliotheken KRNLx86.EXE, USER.EXE, und GDI.EXE, welche eigentlich schon das ganze Windows verkörpern.

In KRNLx86.EXE finden Sie überwiegend Funktionen, die Windows zur System- und Speicherverwaltung benötigt. Dagegen beinhaltet USER.EXE hauptsächlich Funktionen zur Darstellung und Verwaltung von Fenstern oder Menüs.

☞ Sie wissen ja, dass Windows jeden Button, jedes Steuerelement oder Kontrollfeld allgemein als Fenster verwaltet. In der dritten Systemlibrary GDI.EXE finden Sie Graphik und Ausgabefunktionen für Fenster.

Nun wollen wir uns erst einmal primär mit folgender Frage beschäftigen: Warum lassen sich API-Funktionen ohne weiteres innerhalb unseres Programms aufrufen?

Solange es sich bei den benötigten API-Aufrufen um die erwähnten Hauptbibliotheken handelt, reicht es aus, wenn Sie die Unit `Windows` in Ihr Programm einbinden, da in deren Interface-Teil alle benötigten Deklarationen codiert sind. Somit können Sie anschliessend alle Funktionen in Ihrem Programm aufrufen, als wären Sie im Sprachumfang von Delphi enthalten bzw. im Deklarationsteil explizit deklariert.

#### 10.2.2 API-Funktionen nach Gebieten aufgeteilt:

Atom-Management-Funktionen	Mapping-Funktionen
Programmausführungsfunktionen	Speicherverwaltungsfunktionen
32-Bit-Speicherverwaltungsfunktionen	Menüfunktionen
Callback-Funktionen	Botschaftsfunktionen
Caret-Funktionen	Meta-Datei-Funktionen
Zwischenablage	Modulverwaltungsfunktionen
Clipping-Funktionen	OLE-Funktionsgruppen
Kommunikationsfunktionen	Palettenfunktionen
Koordinatenfunktionen	Zeichenstiftfunktionen
Mauszeigerfunktionen	Zeiger-Validierungsfunktionen
DDE-Funktionen	Druckersteuerungsfunktionen

Debug-Funktionen	Eigenschaftsfunktionen
Gerätekontextfunktionen	Rechteckfunktionen
Dialogfensterfunktionen	Regionfunktionen
Anzeigefunktionen	Registrierungsfunktionen
DragDrop	Ressourcenverwaltungsfunktionen
Zeichenwerkzeugfunktionen	Bildlauffunktionen
Ellipsen- und Polygon-Funktionen	Segment-Funktionen
Fehler-Funktionen	Shell-Funktionen
Datei-I/O-Funktionen	Stress-Funktionen
Schrift-Funktionen	String-Manipulationsfunktionen
GDI-Funktionen	Systemfunktionen
Hardware-Funktionen	Task-Funktionen
Hook-Funktionen	Text-Funktionen
Symbol-Funktionen	Toolhelp-Funktionen
Informationsfunktionen	TrueType-Funktionen
Initialisierungsdatei-Funktionen	User-Funktionen
Version-Funktionen	Kernel-Funktionen

### 10.2.3 Windows-Messages

Ein Kernpunkt der traditionellen Windows-Programmierung liegt in der Behandlung der Botschaften, die Windows an Anwendungen sendet. Delphi behandelt beinahe alle gängigen Botschaften für Sie. Bei einer flexiblen und unabhängigen Entwicklung kann es jedoch vorkommen, dass Sie Botschaften erzeugen müssen, die Delphi noch nicht behandelt, oder dass Sie Ihre eigenen Botschaften erzeugen, die natürlich anschliessend auch von Delphi behandelt werden müssen.

Drei wichtige Aspekte sind bei der Arbeit mit Botschaften zu beachten:

- Das Botschaftsbehandlungssystem im Detail
- Verändern der Botschaftsbehandlung
- Erweitern neuer Botschaftsbehandlungsroutinen

Alle Delphi-Objekte besitzen eingebaute Mechanismen zur Behandlung von Botschaften, die man **Botschaftsbehandlungsmethoden** oder eben Event Handler nennt. Das

- grundlegende Konzept von Botschaftsbehandlungsroutinen
- ☞ besteht nun darin, dass eine Fensterfunktion im Objekt wie auch immer geartete Botschaften empfängt und sie zuteilt. Den fortlaufenden **Event Loop** mit den drei API-Funktionen `getMessage`, `translateMessage` und `dispatchMessage` erledigt Delphi völlig automatisch im Hintergrund.
  - ☞ Werfen Sie einmal einen Blick auf die `Application.Run` Methode in der Projektdatei. Diese Methode ist der Startpunkt der ganzen Komplexität. Dabei wird, abhängig von der empfangenen Botschaft eine Methode aus einer angegebenen Auswahl aufgerufen. Ob ein Event Handler vorhanden ist oder nicht überprüft in Delphi der interne **Dispatcher**, nicht zu verwechseln mit dem Windows Dispatcher, der eine API-Funktion darstellt. Existiert für eine bestimmte Botschaft keine besondere Methode, wird die Standardbehandlungsroutine **Default Handler** ausgeführt. In der klassischen Windows-Programmierung benötigte man früher noch umfangreiche Case-Strukturen mit Einträgen für jede Botschaft.
- Das folgende Schema illustriert das Botschaftszuteilungssystem und präsentiert zugleich die komplexe Nachrichtenarchitektur unter Windows:

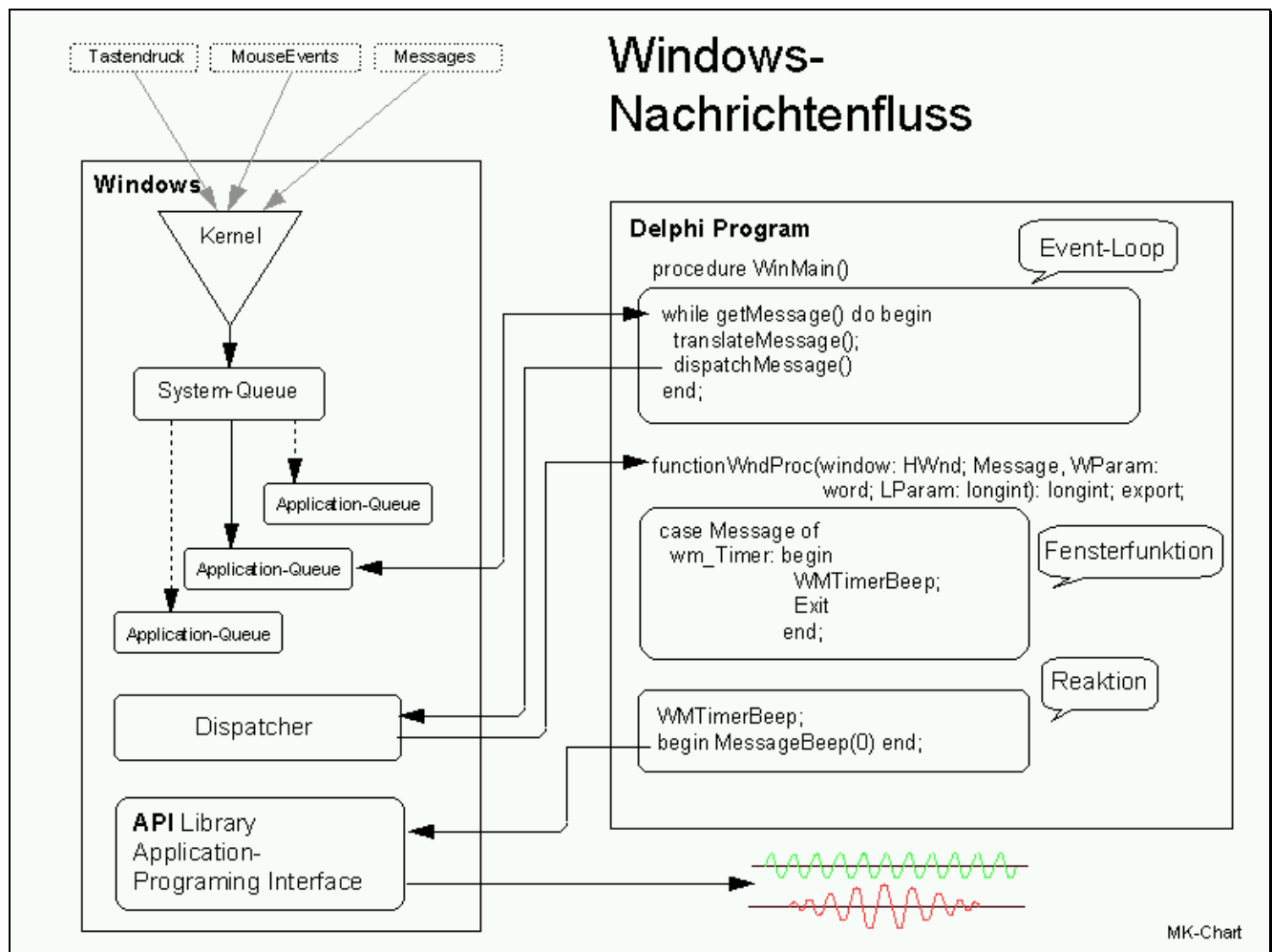



Abb. 26 Übersicht über den Windows-Nachrichtenfluss

Die Delphi-Komponentenbibliothek definiert ein **Botschaftszuteilungssystem**, das alle Windows-Botschaften (einschliesslich benutzerdefinierter Botschaften), die man an ein bestimmtes Objekt leitet, in Methodenaufrufe übersetzt. Es sollte niemals (never say never) notwendig sein, dass Sie dieses Botschaftszuteilungssystem ändern müssen. Ihre Aufgabe besteht nur darin, adäquate Botschaftsbearbeitungsmethoden zu erzeugen, d.h. die Reaktion auf die Nachricht zu bestimmen.

Der grösste Vorteil dieses Botschaftszuteilungssystems liegt jedoch darin, **dass Sie jede beliebige Botschaft an jede beliebige Komponente zu jeder beliebigen Zeit senden können**. Besitzt die Komponente keine für die Botschaft definierte Behandlungsroutine, wird die Standardbehandlung wirksam.

#### 10.2.4 Was enthält eine Windows-Botschaft?

Eine Windows-Botschaft stellt einen Daten-Record vom Typ `TMessage` dar, der verschiedene nützliche Felder enthält. Das wichtigste Feld ist ein Integer-Wert, der die Botschaft identifiziert. Windows definiert sehr viele Botschaften, und die Unit `Messages` deklariert Bezeichner für jede dieser Botschaften. Weitere nützliche Informationen einer Nachricht befinden sich in zwei Parameterfeldern und in einem Ergebnisfeld. Der Windows-Programmcode bezieht sich häufig auf diese Werte als `wParam` und `lParam`, also „word parameter“ und „long parameter“ (siehe Abb. 26). In vielen Fällen enthält jeder Parameter mehr als nur eine Information.

 Ursprünglich mussten Windows-Programmierer auswendig wissen (oder nachschlagen), was jeder Parameter enthält. Microsoft hat nun die Parameter mit Namen versehen. Dieses sogenannte „**Extrahieren von Botschaften**“ macht es nun viel einfacher, die mit jeder Botschaft verknüpften Informationen zu verstehen. Beispielsweise heißen die Parameter der Botschaft `WM_KEYDOWN` jetzt `VKey` und `KeyData`. Die hierdurch zur Verfügung gestellte Information ist sehr viel genauer als in `wParam` und `lParam`.

Delphi definiert einen besonderen Recordtyp für jeden Botschaftstyp, der jeden Parameter mit einem mnemonischen Namen versieht. Mit dem Mausbotschafts-Record z.B. müssen Sie sich nicht mehr darum kümmern, welche Koordinate in welches Datenwort gehört, da Sie sich über die Namen `XPos` und `YPos` auf die Parameter beziehen können.

#### 10.2.5 Eigene Botschaften versenden

Bei unseren bisherigen Betrachtungen haben wir nur die eingehenden Botschaften bearbeitet. Delphi kann sich jedoch auch aktiv am Nachrichtenverkehr beteiligen, indem Sie eigene Botschaften versenden. Ein Beispiel gefällt?

```
Procedure frmEditor.BtnScrSaveClick(Sender:
                                TObject);
begin
    PostMessage(hWnd_BroadCast, wm_SysCommand,
                sc_ScreenSave, 0)
end;
```



Testen Sie diese Aktion kurz im Editor Beispiel.

Ihnen ist sicher die API-Funktion `PostMessage` aufgefallen zu der sich noch `SendMessage` gesellt. Hier der Unterschied: Die Funktion `SendMessage` wartet, bis die Botschaft bearbeitet ist, und kehrt erst dann zurück. Im Gegensatz dazu legt `PostMessage` die Botschaft einfach in die **Application Queue** und kehrt sofort zurück, d.h. die Nachricht wird ev. später bearbeitet. Wenn Sie also darauf bestehen, die Nachricht sofort zu bearbeiten, brauchen Sie `SendMessage`, wobei dann Behutsamkeit angesagt ist.

#### 10.2.6 Überschreiben von Botschaften

Natürlich ist es auch möglich, Botschaften abzufangen und entsprechend eigene Methode zu definieren. Zum Überschreiben einer Botschaftsbearbeitungsroutine deklarieren Sie eine neue Methode mit dem gleichen Botschaftsindex wie in der Methode, die überschrieben wird. Sie müssen den Befehl `message` und einen passenden **Botschaftsindex** verwenden.



Beachten Sie, dass der Name der Methode und der Typ des einzelnen var-Parameters nicht zu der überschriebenen Methode passen müssen. Nur der Botschaftsindex ist ausschlaggebend. Es dient jedoch der Klarheit, wenn Sie die Konventionen der Namensgebung für Botschaftsbearbeitungsmethoden beachten, die sich an den betreffenden Botschaften orientieren. Um beispielsweise die Behandlung der `WM_PAINT`-Botschaft zu überschreiben, deklarieren Sie die Methode `WM_PAINT` neu.

```
type TMyComponent = class(...)
public
    procedure WMPaint(var Message: TWMPaint);
                           message WM_PAINT;
end;
```

#### 10.3 Definieren eigener Botschaften

Eine Vielzahl von Standardkomponenten definiert Botschaften für den internen Gebrauch. Die häufigsten Gründe für das Definieren von Botschaften liegen in Rundrufinformationen (**Broadcast-Messages**), welche die Windows-Standardbotschaften nicht abdecken und Benachrichtigungen über Statuswechsel oder Statusinformationen anzeigen.

### 10.3.1 Deklarieren eines eigenen Botschaftsbezeichners

Ein Botschaftsbezeichner ist eine Integerwert-Konstante. Windows reserviert die Botschaften unterhalb 1024 für seinen eigenen Gebrauch. Wenn Sie Ihre eigenen Botschaften deklarieren, sollten Sie also erst oberhalb dieser Ebene beginnen. Die Konstante `WM_USER` stellt die Anfangszahl für benutzerdefinierte Botschaften dar. Wenn Sie Botschaftsbezeichner definieren, sollten diese auf `WM_USER` basieren.

☝ Denken Sie daran, dass einige Windows-Standarddialogelemente Botschaften im benutzerdefinierten Bereich verwenden. Dies betrifft Listenfenster, Kombinationsfenster, Editierfenster und Befehlsschalter. Wenn Sie aus einem dieser Elemente eine Komponente ableiten und für diese eine neue Botschaft definieren wollen, müssen Sie die Unit `Messages` überprüfen, damit Sie wissen, welche Botschaften Windows für das betreffende Dialogelement bereits reserviert hat.

Als Beispiel sehen Sie 4 benutzerdefinierte Botschaften:

```
const    WM_MYFIRSTMSG = WM_USER + 0;
          WM_MYSECONDMSG = WM_USER + 1;
          CM_LADEN = CM_FIRST + 100;
          CM_SPEICHERN = CM_FIRST + 101;
```

Solche benutzerdefinierten Botschaften dienten früher dazu, auf die Auswahl eines Menüs reagieren zu können. Ich kann mich noch erinnern, als wir zum ersten Mal den Begriff Message-Response-Methoden hörten und glaubten, in einem Marketingseminar zu stecken. Hier noch ein vollständiges Beispiel einer benutzerdefinierten Nachricht, welche z.B. für eine Aktualisierung erhalten könnte :

```
1  const
2      MY_UPDATE = WM_USER + 0;
3  type
4      TForm1 = class(TForm)
5      ...
6          procedure memEditKeyDown(...);
7      protected
8          procedure MYUpdate(var message:
9              TMessage); message MY_UPDATE;
10     ...
11     end;
12
13 implementation
14
```



```

15 procedure TForm1.memEditKeyDown(...);
16 begin
17     PostMessage(Handle, MY_Update, 0, 0);
18     {eigene message auf queue legen}
19 end;
20
21 procedure TFrmEditor.MYUpdate(var message :
22                               TMessage);
23 begin
24     do something
25 end

```

### 10.3.2 Generelle Botschaftsbearbeitung

Das zentrale Ereignis `OnMessage` tritt ein, wenn Ihre Anwendung eine Windows-Nachricht empfängt. Durch Erzeugen einer Ereignisbehandlungsroutine vom Typ `OnMessage` in Ihrer Anwendung können Sie andere Handler aufrufen, die auf die Nachricht reagieren. Eine Ereignisbehandlungsroutine vom Typ `OnMessage` lässt Ihre Anwendung die Nachricht abfangen, bevor Windows selbst diese verarbeitet.

Das folgende Beispiel zeigt die Uhrzeit als serielle Zahl der gerade empfangenen Windows-Nachricht in der Beschriftung `Caption` von `lblTime` an. `CatchMessage` sollte als eine Methode von `TFrmEditor` deklariert sein. Was das Beispiel so interessant macht ist folgendes:

Wenn Sie sehr schnell über der Applikation die Maus bewegen, spielen Sie sozusagen Nachrichtenabfangjäger und simulieren so einen Timer, der die Uhrzeit ständig aktualisiert.

```

1 procedure TFrmEditor.FormCreate(Sender:
2                               TObject);
3 begin
4     Application.OnMessage:= CatchMessage;
5 end;
6
7 procedure TFrmEditor.CatchMessage(var Msg:
8                                   TMsg; var Handled:Boolean);
9 begin
10    lblTime.Caption:= IntToStr(Msg.Time);
11 end;

```

## **10.4 Rekapitulation**

### **10.4.1 Zusammenfassung**

Properties sind die augenfälligsten Teile von Komponenten und zugleich diejenigen, die Anwendern und Entwicklern, die mit den Komponenten arbeiten, den grössten Nutzen bringen.

Die Deklaration eines Property enthält drei Dinge:

- Den Namen des Property
- Den Typ des Property
- Methoden für das Lesen und/oder Schreiben des Property-Wertes

Die Reaktion auf Windows-Botschaften stellt für einige Komponenten ein wichtiges Verfahren dar, mit Windows oder mit anderen Komponenten zu interagieren. Delphis Standardkomponenten reagieren bereits auf die gängigsten Botschaften. Ihre Komponenten können jedoch diese Behandlung überschreiben oder auf andere Botschaften, inklusive benutzerdefinierte, reagieren.

Ein Kernpunkt der traditionellen Windows-Programmierung liegt in der Behandlung der Botschaften, die Windows an Anwendungen sendet. Delphi behandelt beinahe alle gängigen Botschaften automatisch.

### **10.4.2 Kapiteltest**

1. Wahr oder Falsch? Die Deklaration eines Properties allein alloziert keinen Speicher für den Property-Wert.
2. Wie viele Parameter benötigt eine Read-Methode?
3. Wie viele Parameter benötigt eine Write-Methode?
4. Welche Gruppe von API-Funktionen sind für geräteunabhängige Grafiken (Device Independent Bitmaps) verantwortlich?
5. Welche 6 Parameter beinhaltet ein Nachrichten-Record in Delphi?

### **10.4.3 Übung**


Senden Sie an einen Button Ihrer Wahl die Nachricht, das er gedrückt wird.

```
PostMessage(Button1.Handle, cn_Command,  
            bn_Clicked, 0);
```

Überschreiben Sie die schon erwähnte Botschaft `WM_PAINT` mit einer Methode welche als Reaktion ein `Message-Beep(0)` von sich gibt. Sie werden staunen wie oft Windows diese Botschaft aktiviert.


 **Vergessen Sie auf gar keinen Fall die Vorfahr-Methode mit `inherited` aufzurufen.**


## 11 Exception Handling und RTTI

 In diesem Teil werden Sie in die Kunst der defensiven sprich sicheren und robusten Programmierung eingeführt.

- Sie lernen auf Ausnahmen zu reagieren,
- anwenderfreundliche Fehlermeldungen zu definieren
- und mit Laufzeitinformationen umzugehen.

### 11.1 Was sind Ausnahmebehandlungen?

 Auch mit der **Ausnahmebehandlung** folgt Delphi bekannten Konzepten anderer OOP-Sprachen. **Exceptions** sind Objekte, die Informationen über Fehlerzustände enthalten, derentwegen ein Programm eine Operation nicht wie geplant ausführen konnte; zum Beispiel aus Speichermangel, wegen einer Schutzverletzung oder einer Division durch Null. Trat bisher eine derartige Bedingung auf, brach das Programm mit einem Laufzeitfehler ab. Um dies zu vermeiden, benötigte man immer tiefer geschachtelte IF-Statements mit alternativen Ausführungspfaden. Exceptions erlauben statt dessen gebündelte Fehlerbehandlung.

 Eine **Exception** ist ganz allgemein eine Fehlerbedingung oder ein anderes Ereignis, das den normalen Ablauf einer Anwendung unterbricht. Wenn eine Exception ausgelöst wird, geht die Programmsteuerung von der gerade ausgeführten Anweisung auf eine Exception-Behandlungsroutine über. Object Pascal unterstützt die Ausnahmebehandlung durch die Bereitstellung einer Struktur, mit welcher der Compiler die normale Programmlogik von der Behandlung der Ausnahmefälle trennt. Dies führt zu besser wartbaren und erheblich robusteren Anwendungen.

Object Pascal stellt Exceptions durch Objekte dar. Dies hat eine Reihe von Vorteilen, zu denen als wichtigste zählen:


- Exceptions lassen sich durch Vererbung hierarchisch gruppieren.
- Neue Exceptions sind verwendbar, ohne bestehenden Code zu beeinflussen.
- Ein Exception-Objekt kann Informationen (wie z.B. eine Fehlermeldung oder einen Fehlercode) vom Ort der Auslösung bis zum Ort der Behandlung transportieren.



```

11     AnInteger := 10 div ADiv; { dies erzeugt
12                               einen Fehler }
13     finally
14         FreeMem(APointer, 1024); {die
15     Ausführung wird trotz des Fehlers hier
16     wieder aufgenommen }
17     end;
18 end;

```


 In typischen Anwendungen werden try...finally-Anweisungen immer häufiger eingesetzt als try...except-Anweisungen. Beispielsweise verlassen sich Entwickler, die die **VCL** verwenden, normalerweise auf deren standardmässige Exception-Behandlung, so dass man try...except-Anweisungen hier kaum noch benötigt.


Die folgenden Ressourcen sollten Sie zum Beispiel immer freigeben

- Dateien
- Speicher
- Windows-Ressourcen
- Objekte

#### 11.1.2 try except Construct

Nun schreiten wir zu dem, was bisher verschiedenen Präsidenten und Regierungen vorbehalten war: **Ausnahmezustände** verhängen. Mit den im letzten Abschnitt beschriebenen Aktionen ist der Ausnahmezustand jedoch noch nicht aufgehoben. Nach der Ausführung des finally-Blocks wird das Exception-Objekt so lange eine Aufrufebene höher gereicht, bis ein except-Block zur Verfügung steht.

 Eine **Exception-Behandlungsroutine** besteht aus Code, der eine oder mehrere bestimmte Exceptions behandelt, die innerhalb eines geschützten Code-Blocks auftreten. Um eine Exception-Behandlungsroutine zu definieren, betten Sie den Code, den Sie schützen wollen, in einen Block ein und geben die Anweisungen zur Behandlung der Exception im Abschnitt except des Blocks an.

 Bleiben wir weiterhin bei unserer unglücklichen Division durch Null. Sie können eine Exception-Behandlungsroutine für die Division durch Null definieren, um ein Standardergebnis zu erhalten:

```

1 function GetAverage(Sum,NumberOfItems:
2                      Integer): Integer;

```

```

3  begin
4      try
5          Result := Sum div NumberOfItems;
6      except
7          on EDivByZero do Result:= 0;
8      end;
9  end;

```

Dies ist viel klarer, als bei jedem Aufruf der Funktion auf Null testen zu müssen. Durch die Verwendung von Exceptions können Sie den „normalen“ Ausdruck Ihres Algorithmus festlegen und für die Ausnahmefälle vorsorgen. Ohne Exceptions müssen Sie jedesmal untersuchen, ob Sie den nächsten Schritt in der Berechnung auch wirklich ausführen dürfen.

#### 11.1.3 Nochmaliges Auslösen einer Exception

Wenn Ihre lokale Behandlungsroutine die Behandlung abgeschlossen hat, entfernt sie natürlich die Exception-Instanz, und die Behandlungsroutine des umschließenden Blocks kann nicht mehr aktiv werden.

Wenn eine Exception auftritt, wollen Sie für den Anwender wahrscheinlich eine Meldung ausgeben und anschliessend mit der Standardbehandlung fortfahren. Hierzu definieren Sie eine lokale Exception-Behandlungsroutine, die die Meldung anzeigt und danach aufruft. Man nennt dies die Exception nochmals auslösen. Nach diesem Muster können Sie im Prinzip **jede Ausnahmebedingung** abfangen. Das folgende Beispiel illustriert dieses Verfahren:

```

1  procedure TForm1.Button4Click(Sender:
2      TObject);
3  var realFloat: real;
4  begin
5      try
6          { Anweisungen }
7          try
8              { besondere Anweisungen }
9              realFloat:= realFloat / 0;
10         except
11             on EZeroDivide do
12                 begin
13                     { Behandlung nur der besonderen
14                       Anweisungen }
15                     MessageDlg('EZeroDivideist ausgelöst!',
16                               mtError, [mbOK], 0);

```

```

17         raise    { nochmaliges Auslösen der
18                   Exception }
19         EGPFault.Create('Hier spielen wir'
20         +' Schutzverletzung!');
21     end;
22 end;
23 except
24     on EZeroDivide do    { für alle Fälle }
25     messagebeep(0);
26 end;
27 end;

```



Beachten Sie, dass es zwei NullDivision-Exception gibt:

EDivByZero, Vorgänger: EIntError, Int durch Null

EZeroDivide, Vorgänger: EMathError, Float durch Null

#### 11.1.4 Verschachtelte Exceptions

Im Code einer Exception-Behandlungsroutine lassen sich wiederum andere Exceptions auslösen und behandeln. Solange die Exceptions, die in einer Exception-Behandlungsroutine ausgelöst werden, auch dort behandelt werden, wirkt sich dies auf die ursprüngliche Exception nicht aus. Wenn sich die Reichweite einer solchen Exception jedoch so vergrößert, dass sie über diese Behandlungsroutine hinausgeht, geht die ursprüngliche Exception verloren. Das obige Beispiel konkretisiert auch diesen Sachverhalt.

#### 11.1.5 Die vier Ausnahmestände im Überblick

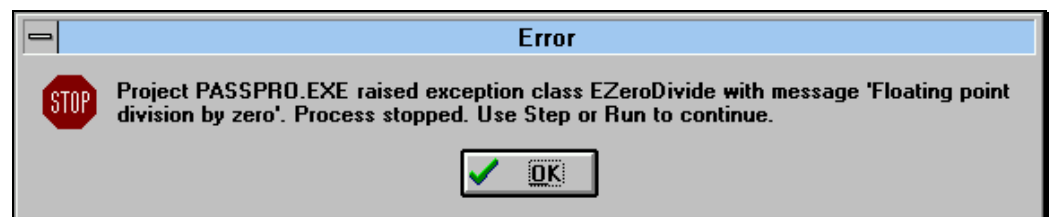


Abb. 27 Der integrierte Debugger hat die Anwendung unterbrochen



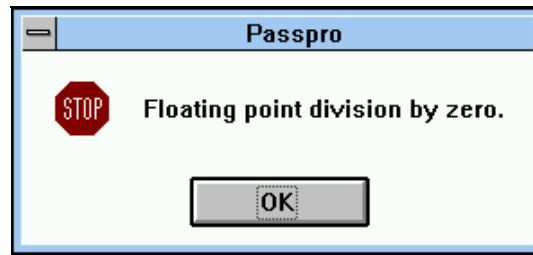


Abb. 28 Der integrierte Debugger hat die Anwendung nicht unterbrochen, die Ausnahme wurde direkt an die Laufzeitbibliothek weitergeleitet.

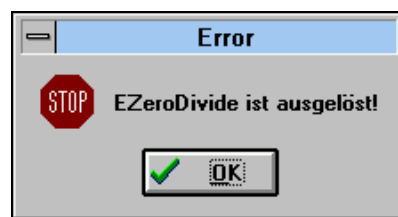


Abb. 29 Abfangen der Exception mit eigenem Dialog

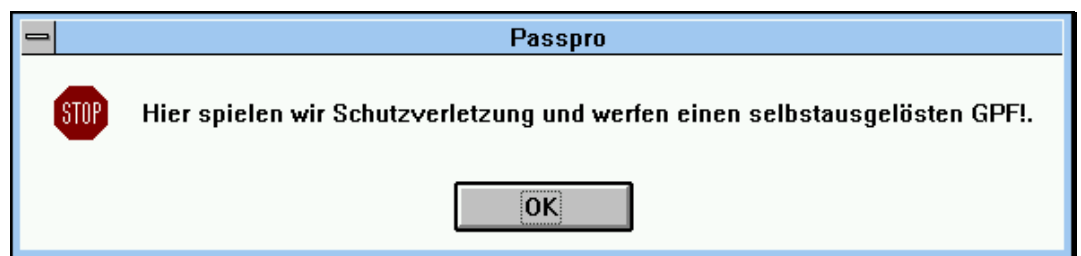


Abb. 30 Eigene Exception mit eigenem Fehlertext

☞ Gemäss einer Statistik gehört die Division durch Null immer noch zu den häufigsten Fehlern. Kurzer Blick ins Delphi-Handbuch zeigt folgendes Bild:

Schwerwiegende Fehler. Diese Fehler führen in jedem Fall zum sofortigen Programmabbruch.

200 Division by zero (Division durch Null)

Das Programm hat versucht, während einer /-, mod- oder div-Operation eine Zahl durch Null zu dividieren.

## 11.2 Die Exception-Klasse

Eigene Exceptions zu erzeugen ist natürlich möglich. Dazu reicht es schon aus, eigene Referenzen von bestehenden Exceptions zu erzeugen:

Type

```
MyException = Class(EMathError);
```



Dabei bildet die Klasse `Exception` die Mutter aller in Delphi existierenden Exceptions. Suchen Sie einmal im Browser unter `SEARCH` und `BROWSE SYMBOL` nach **exception**. Sie sehen dann die gesamte Exception-Hierarchie.

Nun haben Sie zwar auch einen Nachfolger von `Exception` gebildet, aber `MyException` ist gleichzeitig auch als eigene Klasse definiert. Wir können also `MyException` ganz im Sinne der Theorie der objektorientierten Programmierung erweitern. Sinnvolles Beispiel dafür ist die in Delphi bestehende Exception-Klasse `EInOutError`:

Type

```
EInOutError = Class(Exception);
    ErrorCode: Integer;
end;
```

Neben allen Elementen der Klasse `Exception` definiert `EInOutError` auch noch das Feld `ErrorCode`, das die Fehlernummer der fehlgeschlagenen Ein-/Ausgabeoperation unter DOS enthält.

### 11.3 Run Time Type Information

#### (Typinformation zur Laufzeit)

Diese aus der C++ Welt ererbte Gut bringt eine enorme Flexibilisierung mit sich. Delphi gibt den Programmen zur Laufzeit Zugriff auf Informationen über den Objekttyp. Insbesondere kann man den neuen Operator `is` benutzen, um zu bestimmen, ob ein gegebenes Objekt von einem gegebenen Typ oder einer seiner Nachkommen ist.

#### 11.3.1 Der Operator `is`



Der Operator `is` ermöglicht **dynamische Typüberprüfungen**. Mit dem `is`-Operator können Sie überprüfen, ob der aktuelle (zur Laufzeit vorliegende) Typ einer Objektreferenz zu einer bestimmten Klasse gehört. Die Syntax des `is`-Operators lautet wie folgt:


```
ObjectRef is ClassRef
```

Dabei ist `ObjectRef` eine **Objektreferenz** und `ClassRef` eine **Klassenreferenz**. Der `is`-Operator liefert einen Booleschen Wert. Das Ergebnis ist `True`, wenn `ObjectRef` eine

Instanz der über `ClassRef` angegebenen Klasse oder einer davon abgeleiteten Klasse ist. Andernfalls ist das Ergebnis `False`. Enthält `ObjectRef` den Wert `nil`, ist das Ergebnis immer `False`. Der Compiler meldet den Fehler „Typen nicht miteinander vereinbar (Type mismatch)“, wenn bekannt ist, dass die deklarierten Typen von `ObjectRef` und `ClassRef` nicht verwandt sind. Dies ist der Fall, wenn der deklarierte Typ von `ObjectRef` weder Vorfahr noch Nachkomme ist.

Der `is`-Operator wird häufig im Zusammenhang mit einer `if`-Anweisung benutzt, um eine **überwachte Typumwandlung** auszuführen, wie z.B.:

```
if ActiveControl is TEdit then
    TEdit(ActiveControl).SelectAll;
```

 Wenn die `is`-Abfrage den Wert `True` liefert, kann `ActiveControl` sicher in die Klasse `TEdit` umgewandelt werden. Wenn man den `is` Operator mit anderen boolschen Ausdrücken mixt muss man den `is`-Test in Klammern setzen:

```
if (Sender is TButton) and
   (TButton(Sender).Tag <> 0) then ...;
```

### 11.3.2 Der Operator `as`

Der Operator `as` ermöglicht **überprüfte Typumwandlungen**. Die Syntax des `as`-Operators lautet wie folgt:

```
ObjectRef as ClassRef
```

Ergebnis ist eine Referenz auf das Objekt, auf das `ObjectRef` verweist, jedoch mit dem Typ, auf den `ClassRef` verweist. Zur Laufzeit muss `ObjectRef` den Wert `nil` enthalten oder eine Instanz der über `ClassRef` angegebenen Klasse oder einer davon abgeleiteten Klasse sein. Ist keine dieser Bedingungen erfüllt, wird eine Exception ausgelöst. Ansonsten ähnliches Verhalten wie der `is`-Operator.

Der `as`-Operator wird häufig im Zusammenhang mit einer `with`-Anweisung benutzt, wie z.B.:

```
with Sender as TButton do begin
    Caption : = '&OK';
    OnClick : = OkClick;
```

end;



Wird der `as`-Operator in einer **Variablenreferenz** benutzt muss die `as`-Typenumwandlung in Klammern eingeschlossen sein:

```
(Sender as TButton).Caption := '&Ok';
```

## 11.4 Klassenreferenzen

### 11.4.1 Klassenreferenztypen



**Klassenreferenztypen** ermöglichen die direkte Durchführung von Operationen auf Klassen. Dies unterscheidet sie von Klassentypen, bei denen Operationen auf **Instanzen** von Klassen durchgeführt werden. Klassenreferenztypen werden manchmal auch als **Metaklassen** oder **Metaklassentypen** bezeichnet.

class reference type

class object type identifier class of

Klassenreferenztypen sind in folgenden Zusammenhängen vorteilhaft:

- zusammen mit einem virtuellen Konstruktor zum Erzeugen eines Objekts, dessen Typ zum Zeitpunkt der Compilierung noch nicht bekannt ist
- zusammen mit einer Klassenmethode zum Durchführen einer Operation auf einer Klasse, dessen Typ zum Zeitpunkt der Compilierung noch nicht bekannt ist
- als Operand auf der rechten Seite eines `is`-Operators zum Durchführen einer dynamischen Typüberprüfung eines Typs, der zum Zeitpunkt der Compilierung noch nicht bekannt ist
- als Operand auf der rechten Seite eines `as`-Operators zum Durchführen einer überprüften Typumwandlung eines Typs, der zum Zeitpunkt der Compilierung noch nicht bekannt ist.

## 11.5 Rekapitulation

### 11.5.1 Zusammenfassung

Exceptions sind Objekte, die Informationen über Fehlerzustände enthalten, derentwegen ein Programm eine Operation nicht wie geplant ausführen konnte; zum Beispiel aus Speichermangel, wegen einer Schutzverletzung oder einer Division durch Null. Trat bisher eine derartige Bedingung auf, brach das Programm mit einem Laufzeitfehler ab. Um dies zu vermeiden, benötigte man immer tiefer geschachtelte IF-Statements mit alternativen Ausführungspfaden. Exceptions erlauben statt dessen gebündelte Fehlerbehandlung.

☞ Manchmal ist es notwendig, hilfreich oder auch nur sicherer, wenn man zur Laufzeit abfragen kann, welchen Typ ein Objekt hat. Auch beim Konzept der Polymorphie ist es äusserst hilfreich zu wissen, welche Klasse sich hinter dem vielgestaltigen Objekt zur Laufzeit tatsächlich verbirgt. Dazu dient die Typinformation zur Laufzeit mit den entsprechenden Klassenreferenzen.

### 11.5.2 Kapiteltest

1. Wahr oder falsch? Die Reihenfolge eines `on do` Statement ist wichtig.
2. Worin unterscheiden sich Klassenreferenzen und Klassentypen?
3. Versuchen Sie einige Hardware-Exceptions der Exception-Klasse zu beschreiben.
4. Wozu dient das folgende erneute Auslösen einer Exception?

```

1 function GetFileList(const Path: string):
2     TStringList;
3 var I: Integer; SearchRec: TSearchRec;
4 begin
5     Result:= TStringList.Create;
6     try
7         I := FindFirst(Path, 0, SearchRec);
8         while I = 0 do begin
9             Result.Add(SearchRec.Name);
10            I := FindNext(SearchRec);
11        end;
12    except
13        Result.Free;
```

```
14         raise;  
15     end;  
16 end;  
17  
1.  
```

### 11.5.3 Übung

Im folgenden Beispiel wird die Exception praktisch an den Benutzer weitergegeben. Erzeugen Sie eine eigene Exception (z.B. *EDateiFehler*) und erweitern dann die Open-Prozedur im Editorbeispiel mit diesem Schutzblock.

```
begin  
    try  
        memEditor.Lines.LoadFromFile....  
        ...  
        btnZeilenNr.Visible := true;  
    except  
        on EFileNotFound do begin  
            MessageDlg('"' + dlgOpenText.FileName  
                + '" lässt sich nicht öffnen',  
                mtInformation, [mbOK], 0); end;  
            raise EDateiFehler.Create(  
                'Hier spielen wir Dateifehler');  
        end;  
    end;  
end;  
end;
```

## 12 Programmstruktur

Auch im Softwareengineering stellt man sich die Frage nach der Lebensdauer einer sterblichen Variablen sprich ihrer Gültigkeit.

- Sie machen Bekanntschaft mit skalierbaren Programmstrukturen
- Sie erfahren die Sichtbarkeiten von Units (Module) und Klassen und
- werden in der Lebenserwartung von statischen, dynamischen und offenen Strukturen geschult, denn Programmieren geht über Studieren

### 12.1 Entwurf der Struktur

Jeden **Bezeichner**, den Sie in Object Pascal verwenden und ihm einen Wert zuweisen oder in einem Ausdruck aufnehmen, müssen Sie vorher deklarieren. Ab dieser Deklaration ist der Bezeichner bis zum Ende seines Gültigkeitsbereiches (Scope) verwendbar.

Beim Entwurf einer Programmstruktur, beachten Sie drei **Gültigkeitsbereichsregeln**:

- Jeder Bezeichner hat nur in dem Block eine Bedeutung, in dem er deklariert wird. Innerhalb des Blocks ist er nach der Stelle bekannt, an dem er deklariert wird.
- Wenn man einen globalen Bezeichner innerhalb eines Blocks neu definiert, erhält die innerste (die am tiefsten geschachtelte) Definition von der Stelle der Deklaration Vorrang bis zum Ende des Blocks.
- Wenn Prozeduren rekursiv aufgerufen werden, zeigt eine Referenz einer globalen Variable immer auf die Instanz dieser Variable im zuletzt erfolgten Prozeduraufruf.

Wir behandeln folgende Arten von Sichtbarkeit:

- Sichtbarkeit von Routinen
- Sichtbarkeit von Blöcken
- Sichtbarkeit von Units
- Sichtbarkeit von Interface- und Standardbezeichnern

- Sichtbarkeit von Klassen

## 12.2 Gültigkeitsbereich von Bezeichnern

☞ Der **Gültigkeitsbereich** eines Bezeichners innerhalb eines Programms, einer oder mehrerer Units gibt an, ob der Bezeichner von anderen Prozeduren und Funktionen im Programm oder der Unit benutzt werden kann oder nicht. Im folgenden sehen wir, wie der Gültigkeitsbereich von Variablen durch verschiedene Strukturen beeinflusst werden kann. Noch kurz eine kleine Entflechtung der Begriffe **Gültigkeit** und **Sichtbarkeit**: Eigentlich spricht man im Zusammenhang von Variablen oder Bezeichnern vom Gültigkeitsbereich welcher sich dann auf die Sichtbarkeit der entsprechenden Struktur (Routine, Klasse, Unit etc. ) ausweitet.

### 12.2.1 Sichtbarkeit von Routinen

Der Gültigkeitsbereich einer Variablen in einer Routine kann entweder lokal oder global sein. Lokale Bezeichner sind nur für die Routinen und Deklarationen sichtbar, die in dem Block enthalten sind, in welchem der Bezeichner deklariert wird. **Globale Bezeichner** werden im Interface-Abschnitt der Unit deklariert und sind für alle Routinen und Deklarationen innerhalb dieser Unit sichtbar.

Hier ein Beispiel zur Abgrenzung lokal versus global:

```

1  program scopeLocalGlobal;
2  var
3      A: integer;    {Globale Variable}
4
5  procedure SetA;
6  var
7      A: integer;    {Erzeugt eine lokale Variable
8                      A}
9  begin
10     A := 4
11 end;                {Zerstört lokale Variable A}
12
13 begin
14     A:= 3;           {Weist globaler Variable A
15                     einen Wert zu}
16     SetA;            {Ruft Prozedur SetA auf}
17     Writeln(A)       {Wert von A = 3 -- nicht 4!}
18 end.
```



### 12.2.2 Sichtbarkeit von Blöcken

In einem **Block** beginnt der Gültigkeitsbereich eines Bezeichners oder Labels am Ort der Deklaration und reicht bis zum Ende des aktuellen Blocks, einschliesslich aller darin geschachtelten Blöcke. Wenn Sie einen Bezeichner innerhalb eines geschachtelten Blocks überschreiben (Neudeklaration), erstreckt sich der Gültigkeitsbereich des neuen Bezeichners nur über den geschachtelten Block und nicht ausserhalb.

Ein Typbezeichner ist nur lokal in dem Block gültig, in dem die Typdeklaration erfolgt. Rekursive Deklarationen sind nur bei Zeigertypen möglich, sonst darf eine Deklaration keine Teile von sich selber enthalten.

Es folgt ein Beispiel mit zwei Blöcken und einer Neudeklaration:

```

1  program Outer;           { Beginn des äusseren
2                           Gültigkeitsbereichs}
3  type
4      TInt = Integer;      { Deklariere I als
5                           Ganzzahltyp }
6  var
7      T: TInt;             { Deklariere T Global }
8  procedure Inner;         { Beginn innerer
9                           Gültigkeitsbereich}
10 type
11     TNew = TInt;          { Neudeklaration: T als
12                           Ganzzahltyp, lokal }
13 var
14     I: TNew;              { Neudeklaration lokal }
15 begin
16     I := 1;
17 end;                      { Ende des inneren
18                           Gültigkeitsbereichs}
19 begin                    {Hauptblock}
20     T := 1;
21 end.                      {Endeäusserer
22                           Gültigkeitsbereich}
23

```

### 12.2.3 Sichtbarkeit von Units

Für den Gültigkeitsbereich von Bezeichnern (Variablen, Prozeduren, Funktionen etc.) die man im **interface**-Abschnitt einer Unit deklariert, gelten dieselben Regeln wie für Blöcke.

Zusätzlich erstreckt sich der Gültigkeitsbereich auf alle Programme die sich der Unit bedienen. Mit anderen Worten, wird ein Bezeichner im **interface**-Abschnitt einer Unit deklariert, ist er für andere Programme und Units verfügbar, die in ihren **uses**-Anweisungen die Unit mit dem betreffenden Bezeichner angeben.

Die erste Unit in der uses-Anweisung stellt den äussersten Gültigkeitsbereich dar, die letzte Unit den innersten Gültigkeitsbereich. Wenn ein Bezeichner in mehreren Units deklariert wird, folgt daraus, dass eine unqualifizierte Referenz auf den Bezeichner diejenige Instanz auswählt, die von der zuletzt aufgeführten Unit in der uses-Anweisung deklariert wurde. Vermeiden Sie also beim Design von Units doppelte Bezeichner. Um eine Instanz auszuwählen, die in einer anderen Unit deklariert wurde, verwenden Sie einen qualifizierten Bezeichner (d.h. einen Unit-Bezeichner mit nachfolgendem Punkt und dem Bezeichner).

Der Gültigkeitsbereich der System-Unit ist global, so dass jedes Programm Zugriff auf die Standardbezeichner von Object Pascal hat. Die System-Unit muss nicht in der uses-Anweisung angegeben werden.

#### 12.2.4 Sichtbarkeit von Interface- und Standardbezeichnern

Programme oder Units, die Uses-Anweisungen enthalten, haben Zugriff auf die Bezeichner, die im Interface-Teil der dort genannten Units deklariert sind. Jede in einer Uses-Anweisung angegebene Unit ergibt einen neuen Gültigkeitsbereich, der die verwendeten Units und das Programm oder die Unit mit der Uses-Anweisung umfasst. Das bedeutet, dass mit jeder zusätzlichen Unit der Gültigkeitsbereich zunimmt:

```
uses
```


```
    SysUtils, WinTypes, WinProcs, Messages,  
    Classes,   Graphics,   Controls,   Forms,  
    Dialogs, StdCtrls;
```

#### 12.2.5 Sichtbarkeit von Klassen

Wenn ein Bezeichner in einer Klassentyp-Deklaration vereinbart wird, erstreckt sich der Gültigkeitsbereich vom Ort der Deklaration bis zum Ende der Klassentyp-Definition sowie über alle Nachkommen des Klassentyps und die Blöcke sämtlicher Methodendeklarationen des Klassentyps.

Die Sichtbarkeit eines Bezeichners (In Klassen lassen sich Eigenschaften, Felder und Methoden deklarieren) wird durch das **Sichtbarkeitsattribut** des Komponentenabschnitts bestimmt, in dem der Bezeichner deklariert ist. Es gibt vier Sichtbarkeitsattribute:

`private`, `protected`, `public` und `published`.

-  Die Schutzebenen `private` und `public` sind allerdings nicht so strikt, wie man zuerst vermuten könnte: Sie gelten nur für den Zugriff aus einer anderen Unit (Modul) heraus. Werden verwandte Klassentypen in der selben Unit untergebracht, so können die Klassentypen **gegenseitig auf ihre privaten** Komponenten zugreifen, als wären sie in C++ als **friend** deklariert.

### ***Private***

Die Sichtbarkeit eines Komponentenbezeichners, der in einem `private`-Abschnitt deklariert wird, ist auf den Modul beschränkt, der die Klassentypdeklaration enthält. Das heisst, `private`-Komponentenbezeichner verhalten sich in dem Modul mit der Klassentypdeklaration wie öffentliche Komponentenbezeichner, ausserhalb des Moduls sind sie jedoch unbekannt, und es ist kein Zugriff auf sie möglich.

### ***Protected***

Die Bezeichner von geschützten Klassenkomponenten und **deren Nachkommen** sind nur sichtbar, wenn der Zugriff über einen Klassentyp erfolgt, der im aktuellen Modul deklariert ist. In allen anderen Fällen sind die Bezeichner geschützter Komponenten verborgen.

Der Zugriff auf die geschützten Komponenten einer Klasse ist nur zur Implementierung von Methoden der Klasse und ihrer Nachkommen möglich. Deshalb werden Klassenkomponenten in der Regel mit `protected` deklariert, wenn sie ausschliesslich bei der Implementierung von abgeleiteten Klassen verwendet werden sollen. Man bezeichnet dies auch als **Entwicklerschnittstelle**.

### ***Public***

Komponentenbezeichner, die man in öffentlichen Abschnitten mit `public` deklariert, unterliegen hinsichtlich ihrer Sichtbarkeit keinen besonderen Einschränkungen.

**Published**

Die Sichtbarkeitsregeln für veröffentlichte Komponenten entsprechen denen für öffentliche Komponenten. Der einzige Unterschied besteht darin, dass für die Felder und Eigenschaften von veröffentlichten Komponenten **Typinformation zur Laufzeit** erzeugt wird (veröffentlichte Komponenten lassen sich mit `published` deklarieren). Die Laufzeitinformationen ermöglichen einer Anwendung die dynamische Abfrage von Feldern und Eigenschaften eines Klassentyps, die andernfalls unbekannt wären.



Die Bibliothek visueller Komponenten (Visual Component Library, VCL) von Delphi greift mit Hilfe der Typinformation zur Laufzeit auf die Werte der Eigenschaften einer Komponente zu, um Formulardateien zu speichern und zu laden. Ausserdem verwendet die Delphi-Entwicklungsumgebung diese Informationen, um die Liste der Komponenteneigenschaften festzulegen, die im Objektinspektor angezeigt wird.

**Sichtbarkeitsattribute im Überblick**

Zugriff auf

<i>Attribut</i>	<i>Klassen Methode</i>	<i>Instanz</i>	<i>Vererbte Klasse</i>
<b>Private</b>	Ja	Nein	Nein
<b>Protected</b>	Ja	Nein	Ja
<b>Public</b>	Ja	Ja	Ja

**12.3 Variablen-Typen****12.3.1 Statische Variablen**

**Typisierte Konstanten** sind mit initialisierten Variablen vergleichbar - das sind Variablen, deren Werte beim Eintritt in den entsprechenden Block definiert werden. Anders als bei untypisierten Konstanten sind bei der Deklaration einer typisierten Konstanten sowohl der Typ als auch der Name der Konstanten vereinbart. Bei einer lokalen Deklaration entspricht eine typisierte Konstante einer statischen Variablen (in C `static`), da sie beim Verlassen des lokalen Bereichs ihren Wert nicht verliert.



### 12.3.2 Dynamische Variablen

Die Bedeutung von Zeigervariablen ist in Object Pascal durch die **implizite Referenzierung** zwar etwas zurückgegangen, aber bei nullterminierten Strings, dynamischen Speicherblöcken ebenso beim Aufbau von dynamischen Datenstrukturen wie Listen, Bäumen und Graphen **sind Zeiger nach wie vor der Eintritt zum effizienten Königreich des Entwicklers**. Ansonsten spezialisiert man sich in Delphi auf den GUI-Bau, Multimedia oder Datenbankanwendungen.

Kleiner Umgang mit Zeigern (siehe auch Kapitel 6.4):

```

1  var
2      ptr : ^string;           {referenzieren}
3  begin
4      New(ptr);                {allozieren}
5      ptr^:= '1001 Nacht';     {dereferenzieren}
6      canvas.TextOut(5,5,ptr^);
7      Dispose(ptr);
8  end;
```

Eine **Referenz** auf die dynamische Variable, auf die eine Zeigervariable zeigt, wird mit Hilfe des Zeigersymbols ^ hinter der Zeigervariablen ausgedrückt.

Dynamische Variablen und ihre Zeigerwerte werden mit den Prozeduren `New` und `GetMem` erzeugt. Der Operator `@` (Adresse-von) und die Funktion `Ptr` erzeugen Zeigerwerte, die man ebenso als Zeiger auf dynamische Variablen behandelt.

`NIL` zeigt auf keine Variable, sondern nirgendwohin (undefiniert). Philosophisch betrachtet ist nirgendwo auch irgendwo.

## 12.4 Offene Parameter

☞ **Offene Parameter** ermöglichen die Übergabe von Strings und Arrays variabler Grösse an die gleiche Prozedur oder Funktion.

### 12.4.1 Offene String-Parameter

Mit `string` deklarierte Parameter sind standardmässig offene String-Parameter. `OpenString` ist kein reserviertes

Wort und lässt sich daher als benutzerdefinierter Bezeichner neu deklarieren. Zudem hat die Bedeutung von Strings in Delphi 2 sowieso eine neue Dimension erhalten (vgl. Kapitel 6.5.2).

Bei offenen String-Parametern können die aktuellen Parameter Variablen eines beliebigen String-Typs sein. In der Prozedur oder Funktion sind die Grössenattribute des formalen und des aktuellen Parameters gleich (d.h. ihre maximale Länge ist identisch).

Offene String-Parameter verhalten sich genauso wie Variablenparameter eines String-Typs, ausser dass sie nicht als normale Variablenparameter an andere Prozeduren oder Funktionen übergeben werden können. Die erneute Übergabe als offene String-Parameter ist jedoch möglich. Im folgenden Beispiel ist der Parameter `s` der Prozedur `StringZuweisen` ein offener String-Parameter:

```
procedure StringZuweisen(var S: OpenString);
begin
    S:= 'Liebe auf den ersten Klick';
end;
```

#### 12.4.2 Offene Array-Parameter

Ein formaler Parameter, der als

array of Type

```
function Summe(const A: array of integer):
    integer;
```

deklariert wird, ist ein **offener Array-Parameter**. Type muss ein Typbezeichner, der aktuelle Parameter eine Variable vom Typ Type oder eine Array-Variable mit Elementen vom Typ Type sein.

Aktueller Parameter der Funktion:

```
Summe(opArray);
```


Auf formale offene Array-Parameter lässt sich nur elementweise zugreifen. Ein offenes Array lässt sich nur als offener Array-Parameter oder untypisierter Variablenparameter an andere Prozeduren und Funktionen übergeben.

☞ Für Wertparameter eines offenen Arrays erzeugt der Compiler eine lokale Kopie des aktuellen Parameters bzw. der aktuellen Funktion auf dem Stack. Bei der Übergabe grosser

offener Arrays als Wertparameter besteht daher **die Gefahr des Stack-Überlaufs**.

Bei Anwendung auf einen offenen Array-Parameter liefert die Standardfunktion `Low` den Wert Null, `High` liefert den Index des letzten Elements des aktuellen Array-Parameters, und die Funktion `SizeOf` liefert die Länge des aktuellen Array-Parameters.

Hier nun das vollständige Beispiel, welches die Summe von 0 bis 100 ermittelt.



```

1  type
2  TOpArray = array[0..100] of integer;
3
4  procedure LoadNumbers(var A: array of
5                        integer);
6  var I: Word;
7  begin
8      for I:= 0 to High(A) do A[I] := I;
9  end;
10
11  function Summe(const A: array of integer):
12                integer;
13  var I: Word; S: integer;
14  begin S:= 0;
15      for I := 0 to High(A) do S:= S +A[I];
16      Summe := S;
17      form1.canvas.textOut(5,5,inttoStr(S));
18  end;
19
20  procedure TForm1.Button6Click(Sender:
21                                TObject);
22  var opArray: TOpArray;
23  begin
24      LoadNumbers(opArray);
25      Summe(opArray);
26  end;
```

Wenn ein Element eines offenen Array-Parameters vom Typ `Char` ist, kann der aktuelle Parameter auch eine String-Konstante sein. Auf der Basis der folgenden Prozedurdeklaration

```

procedure PrintStr(const S: array of Char);
var I: Integer;
begin
```

```
    for I := 0 to High(S) do if S[I] <> #0  
        then Write(S[I]) else Break;  
end;
```

sind z.B. folgende Prozeduraufrufe zulässig:

```
PrintStr('Hello world');  
PrintStr('A');
```

Leere Strings werden bei Übergabe als offenes Zeichen-Array in einen String mit einem einzigen Element konvertiert, das ein Nullzeichen enthält. Die beiden Anweisungen `PrintStr('')` und `PrintStr(#0)` sind daher identisch.

## 12.5 Rekapitulation

### 12.5.1 Zusammenfassung

Der Gültigkeitsbereich eines Bezeichners innerhalb eines Programms, einer oder mehrerer Units gibt an, ob der Bezeichner von anderen Prozeduren und Funktionen im Programm oder der Unit benutzt werden kann oder nicht. Die Sichtbarkeit eines Bezeichners in Klassen (In Klassen lassen sich Eigenschaften, Felder und Methoden deklarieren) wird durch das Sichtbarkeitsattribut bestimmt von denen `private`, `protected`, `public` und `published` zur Verfügung stehen.

Statische und besonders dynamische Variablen sind eine Voraussetzung für offene und flexible Datenstrukturen, welche mit offenen Parametern keine Wünsche offen lassen.

### 12.5.2 Kapiteltest

1. Ermitteln Sie im folgenden Help-Beispiel, was an der Syntax falsch ist?



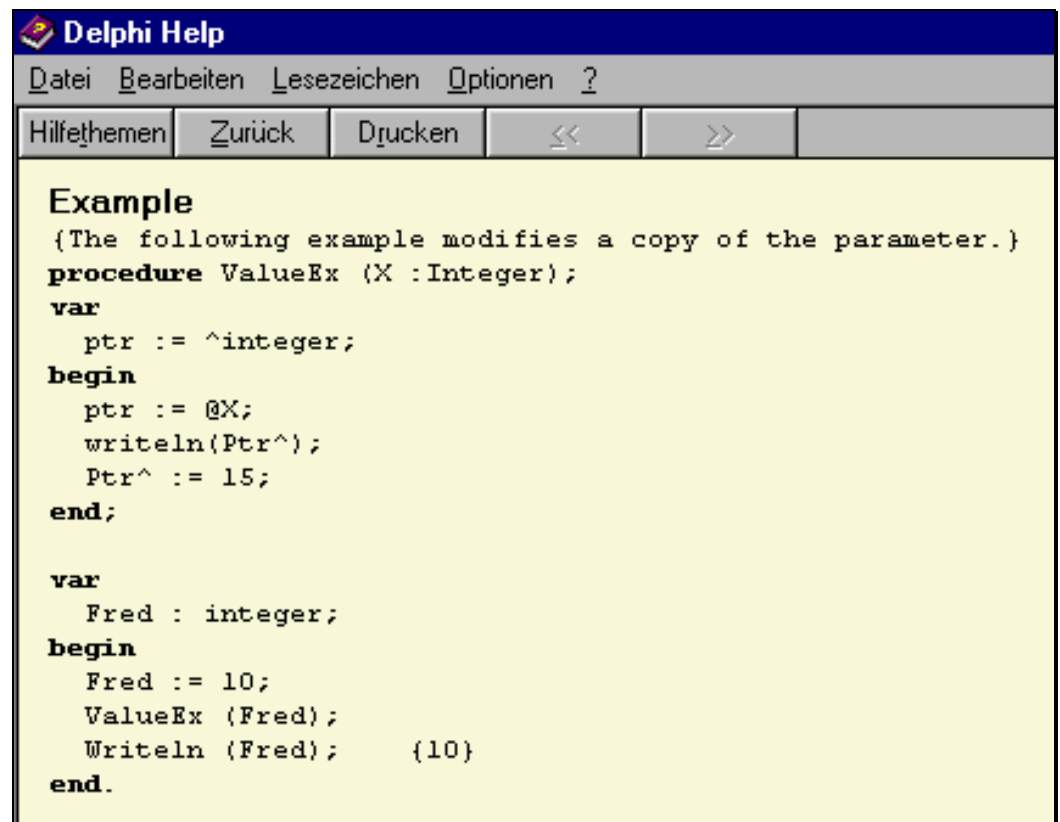


Abb. 31 Auszug Originalhilfe Delphi 2

### 12.5.3 Übung

- 1) Erweitern Sie das Summe 0..100-Beispiel indem Sie mit einem neuen Arraytyp

TAnotherArray = array[0..200] of integer

wieder eine Summe bilden.

### 12.6 Schlussprojekt

Bei unserem Beispiel-Editor fehlt noch als kleine Krönung eine Info-Box (eigenes Formular), welche aus dem Hauptformular aufgerufen wird. Öffnen Sie Ihren Beispiel-Editor und machen Sie Bekanntschaft mit den vorfabrizierten Templates, welche Sie unter FILE - NEW im Register FORMS vorfinden. Wählen Sie die AboutBox und staunen ob der holden Dinge, die da geschehen. Gestalten Sie diese Programm-Visitenkarte nach Belieben. Speichern Sie nun die neu erzeugte Unit unter dem Namen «uAbout.pas». Jetzt geht's ans Eingemachte:

Das Ziel besteht nun darin, die AboutBox aus einem Menüeintrag im Hauptformular zu starten. Ergänzen Sie die Menü Ereignisbehandlung mit folgendem Code:

```
try  
    frmAbout.showModal;  
finally  
    frmAbout.Release;  
end;
```

Wenn Sie alles richtig gemacht haben, sollten Sie etwa das folgende Ergebnis erhalten:



Abb. 32 Der fertige Editor mit einer Info-Box

Falls Sie noch Zeit und Lust haben, können Sie einen weiteren Menüeintrag für die Funktion Ersetzen erstellen und die entsprechende Routine dazu schreiben.

## 13 Begriffsverzeichnis Deutsch - Englisch

Anweisung	statement	Markierungsfelder	check boxes
Anweisungsteil	statement part	Mauspalettenschalter	speed button
Aufzählungstyp	enumerated type	Menü	menu
Ausdruck	expression	Menü-Designer	menu designer
Ausnahmebehandlung	exception handling	Nachkomme	descendant
Ausrichtungspalette	alignment palette	nichtmodales Dialogfenster	modeless dialog box
Bedienfeld	panel	Objekt-Browser	object browser
Bedienfeldkomponente	Panel component	Objektdateien	object files
Bedingte Anweisung	conditional statement	Objektinspektor	object inspector
Besitzer	owner	Objektkennzeichner	object designator
Bezeichner	identifier	öffentlich	public
Bildeditor	image editor	Polymorphie	polymorphism
Bildlaufleiste	scroll bar	privat	private
Bitmap-Schalter	bitmapbutton component	Projektdatei	project file
Blöcke	blocks	Projektoptionen	project options
Botschaft	message	Projektverwaltung	project manager
Codeblock	block	prozeduraler Typ	procedural Type
Compiler-Befehl	compiler directive	qualifizierter	qualified method identifier
Datenbankoberfläche	Database desktop	Methodenidentifizierer	
Datensteuerung	Data Control	Quelltexteditor	code editor
Datenzugriff	data access	Rangfolge der Operatoren	precedence of operators
Deklarationsteil	declaration part	reserviertes Wort	reserved word
Dialogfenster	dialog box	Schaltfeld	button
Eigenschaft	property	Schaltfeldgruppenfeld	radio group box
Eigenschaftsspalte	property column	Schleife	loop
einfacher Typ	simple Type	Schnellhinweis	Hint
Entwicklungsumgebung	development environment	Sichtbarkeit	visibility
Entwurfszeit	design time	spätes Binden	late binding
Ereignis	event	Steuerelement	control
Exception auslösen	to raise an exception	String-Typ	string type
Exception-Behandlungs- routine	exception handler	strukturierte Anweisungen	structured statements
Fehlerbehandlung	error handling	strukturierter Typ	structured Type
Fehlerprüfung	error checking	Symbolleiste	speed bar
Feldkennzeichner	field designator	Tastenkürzel	keyboard shortcuts
Formular	form	Teilbereichstyp	subrange type
Formularschablone	form template	Token	token
Freigabecode	cleanup code	typenumwandelnder Opera- tor	type casting operator
geschützt	protected	typisierte Konstante	typed constant
geschützter Block	protected block	übergeordnetes Element	parent
Gruppenfeldkomponente	GroupBox component	Varianttyp	variant type
Gültigkeitsbereich / Gel- tungsbereich	scope	Vererbung	inheritance
Gültigkeitsbereich von Blöcken	block scope	veröffentlicht	published
Instanz	instance	verschachtelte Eigenschaf- ten	nested properties
integrierter Debugger	integrated debugger	virtuell	virtual
Interface-Teil	interface section	Vorfahr	ancestor
Kapselung	encapsulation	Wertespalte	value column
Klasse	class	Zahl	number
Komponentenbibliothek	component library	Zeichenketten	character strings
Komponentenpalette	component palette	Zeigertyp	pointer type
Konstantendeklaration	constant declaration	Ziehmarke	sizing handle
Laufzeit	run time	zirkuläre Referenz	circular reference
Laufzeit-Typinformation	run-time type information	zusammengesetzte Anwei- sungen	compound statements
lokales Menü des Quelltex- teditors	code editor speed menu	Zuweisungsanweisung	assignment statement
		Zuweisungsoperator	assignment operator

## 14 Glossar

Das folgende Glossar ist ein Auszug aus dem Glossar der Online-Hilfe von Delphi 2.0.

actual parameter	A variable, expression, or constant that is substituted for a formal parameter in a procedure or function call.
ancestor	An object from which another object is derived. An ancestor class can be a parent or a grandparent.
array	A group of data elements identical in type that are arranged in a single data structure and are randomly accessible through an index.
block	The associated declaration and statement parts of a program or subprogram. Examples: In the var block of the routine declare an integer variable. Follow the then of your if..then statement with a begin to start a block of code that will be executed only if the condition is met.
Boolean	A data type that can have a value of either True or False. Data size = byte.
child	<ol style="list-style-type: none"><li>1. A child class is any class that is descended from another. For example, in "type B = class(A)", B is a child of A.</li><li>2. The child of a window appears inside that window and cannot draw outside of its bounds. This is called a child or child window.</li></ol>
class	A list of features representing data and associated code assembled into single entity. A class includes not only features listed in its definition but also features inherited from ancestors. The term "class" is interchangeable with the term "object type."
compiler directive	An instruction to the compiler that is embedded within the program; for example, {\$R+} turns on range checking.
component	<ol style="list-style-type: none"><li>1. The elements of a Delphi application, iconized on the Component palette. Components, including forms, are objects you can manipulate. Forms are components that can contain other components (forms are not iconized on the Component palette).</li><li>2. In Delphi, any class descended from TComponent is, itself, a component. In the broader sense, a component is any class that can be interacted with directly through the Delphi Form Designer. A component should be self-contained and provide access to its features through properties.</li></ol>

constant	An identifier with a fixed value in a program. At compile time, all instances of a constant in source code are replaced by the fixed value.
control	A visual component. Specifically, any descendant of TControl.
descendant	An object derived from another object. A descendant is type compatible with all of its ancestors.
dynamic	A form of virtual method which is more space efficient (but less speed efficient) than simple virtual.
encapsulate	To provide access to one or more features through an interface that protects clients from relying upon or having to know the inner details of the implementation.
exception	An event or condition that, if it occurs, breaks the normal flow of execution. Also, an exception is an object that contains information about what error occurred and where it happened.
expression	Part of a statement that represents a value or can be used to calculate a value.
event	<p>A user action, such as a button click, or a system occurrence such as a preset time interval, recognized by a component.</p> <p>Each component has a list of specific events to which it can respond. Code that is executed when a particular event occurs is called an event handler.</p>
event handler	A form method attached to an event. The event handler executes when that particular event occurs.
field	One possible element of a structured data type (that is, a record or object), a field is an instance of a specific data type. (Compare with property.)
form	To an end user, a form is merely another window. In Delphi, a form is a window that receives components (placed by the programmer at design time, or created dynamically with code at run time), regardless of the intended run-time functionality of the window.
formal parameter	An identifier in a procedure or function declaration heading that represents the arguments that will be passed to the subprogram when it is called.
function	A subroutine that computes and returns a value.

global variable	A variable used by a routine (or the main body of a program) that was not declared by that routine (or a var part of the main body) is considered a global variable by that code. A variable global to one part of a program may be inaccessible to another part of the same program, and hence considered local in that context.
glyph	A bitmap that displays on a BitBtn or SpeedButton component with the component's Glyph property.
header	Text that gives the name of a routine followed by a list of formal parameters, followed in the case of a function by the function's return type. In a unit, a routine may have a header entered into the interface part, and then again in the implementation part. The second appearance of the header may be an exact duplicate of the header in the interface part, or may be only the name of the routine.
identifier	A programmer-defined name for a specific item (a constant, type, variable, procedure, function, unit, program, or field).
implementation	The second, private part of a unit that defines how the elements in the interface part (the public portion) of the unit work.
inheritance	The assumption of the features of one class by another.
instance	A variable of an object type. Actual memory is allocated.
integrated debugger	The integrated debugger is contained within the Integrated Development Environment. This debugger lets you debug your source code without leaving Delphi. The functionality of this debugger can be reached through the Run and View menus.
interface	The first, public part of a unit that describes the constants, types, variables, procedures, and functions that are available within it.
late binding	When the address used to call virtual methods or dynamic methods is determined at run time.
local variable	A variable declared within a procedure or function.
loop	A statement or group of statements that repeat until a specific condition is met.
method	Procedure or function associated with a particular object.

modal	The run-time state of a form designed as a dialog box in which the user must close the form before continuing with the application. A modal dialog box restricts access to all other areas of the application.
modeless	The run-time state of a form designed as a dialog box in which the user can switch focus away from the dialog box without first closing it.
module	A self-contained routine or group of routines. A unit is an example of a module.
nonvisual component	A component that appears at design time as a small picture on the form, but either has no appearance at run time until it is called (like TSaveDialog) or simply has no appearance at all at run time (like TTimer).
object type	A class.
ordinal type	Any Object Pascal type consisting of a closed set of ordered elements.
override	Redefine an object method in a descendant object type.
owner	An object responsible for freeing the resources used by other (owned) objects.
parameter	A variable or value that is passed to a function or procedure.
parent	<ol style="list-style-type: none"> <li>1. The immediate ancestor of a class, as seen in its declaration. Example: In "type B = class(A)", class A is the parent of class B.</li> <li>2. Parent property: the component that provides the context within which a component is displayed.</li> </ol>
pointer	A variable that contains the address of a specific memory location.
private	The keyword indicating the beginning of a class declaration.
procedure	A subprogram that can be called from various parts of a larger program. Unlike a function, a procedure returns no value.
program	<p>An executable file. Less formally, a program and all the files it needs to run.</p> <p>Often used synonymously with 'application'.</p>

project	The complete catalogue of files and resources used in building an application or DLL. More specifically, the main source code file of the programming effort, which lists the units that the application or DLL depends on.
property	A feature that provides controlled access to methods or fields of an object. A published property may also be stored to a file.
protected	Used in class type definitions to make features visible only to the defining class and its descendants.
public	Used in class type definitions to make features visible to clients of that class.
published	Used to make features in class type definitions streamable. Streamable features are visible at design time.
qualified identifier	An identifier that contains a period, that is, includes a qualifier. A qualified identifier forces a particular feature (of an object, record or unit) to be used regardless of other features of the same name that may also be visible within the current scope.
qualifier	An identifier, followed by a period (.), that precedes a method or other identifier to specify a particular symbol reference.
record type	A structured data type that consists of one or more fields.
recursion	A programming technique in which a subroutine calls itself. Use care to ensure that a recursion eventually exits. Otherwise, an infinite recursion will cause a stack fault.
routine	A procedure or function.
scope	The visibility of an identifier to code within a program or unit.
set	A collection of zero or more elements of a certain scalar or subrange base type
statement	The simplest unit in a program; statements are separated by semicolons.
static	Resolved at compile time, as are calls to procedures and methods.
string	A sequence of characters that can be treated as a single unit of data.
type	A description of how data should be stored and accessed. Contrast with variable--the actual storage of the data.



typed constant	A variable that is given a default value upon startup of the application. All global variables occupy a constant space in memory.
unit	<p>A independently compileable code module consisting of a public part (the interface part) and a private part (the implementation part).</p> <p>Every form in Delphi has an associated unit.</p> <p>The source code of a unit is stored in a .PAS file. A unit is compiled into a binary symbol file with a .DCU extension. The link process combines .DCU files into a single .EXE or .DLL file.</p>
value parameter	A procedure or function parameter that is passed by value; that is, the value of a parameter is copied to the local memory used by the routine and therefore, changes made to that parameter are local.
variable parameter	A subroutine parameter that is passed by reference. Changes made to a variable parameter remain in effect after the subroutine has ended.
variable	An identifier that represents an address in memory, the contents of which can change at run time.
virtual	Calls are resolved at run time by name.
visual component	A component that is visible, or can be made visible on a form at run time.

## 15 Index

- &, 68
- (\* \*), 43
- ., 44
- ~\*, 35
- .DCU, 36
- .DFM, 35
- .DLL, 36
- .DPR, 35
- .DSK, 35
- .EXE, 36
- .OPT, 35
- .PAS, 35
- .RES, 35
- :=, 78
- ;;, 44
- { }, 43
- Aktueller Parameter, 179
- ancestor, 58
- AnsiChar, 101
- Anweisungen, 103
- Anweisungsteil, 44
- Anweisungsteil eines Blocks, 104
- Arraytypen, 93
- Aufzählungstypen, 87
- Ausdrücke, 74
- Ausnahmebehandlung, 160
- Ausnahmezustände, 162
- Ausrichtungspalette, 55
- Backup-Dateien, 35
- Bäume, 177
- Bedingte Anweisungen, 105
- benutzerdefinierte Typen, 86, 92
- Bezeichner, 27, 32, 76, 171
- Bilddateien, 36
- Bildeditor, 25, 36
- binär, 107
- binäre Operatoren, 73
- Bitmap-Schalter, 40
- Block, 104, 173
- boolean, 86
- boolesche Logik, 107
- boolescher Typ, 85
- Botschaftsbehandlungsmethoden, 152
- Botschaftsindex, 155
- Botschaftszuteilungssystem, 153
- Broadcast-Messages, 155
- byte, 85
- ByteBool, 86
- call by reference, 116
- call by value, 116
- Caption, 27
- cardinal, 85
- Case-Anweisung, 108
- Char, 86
- child, 58
- chr(13), 91
- Compiler-Befehl, 43
- Compiler-Ressourcen-Datei, 35
- Container-Komponente, 56
- Daten, 72
- Debugging, 37
- default, 147
- Default Handler, 152
- Deklaration einer Methode, 134
- Deklarationsteil, 44, 74
- Deklarationsteil eines Blocks, 104
- descendant, 58
- Desktop-Einstellungen, 35
- Destruktor, 133, 135
- Dialogfenster Alignment, 55
- DLL, 150
- Dynamische Methode, 128
- dynamische Typüberprüfungen, 166

- Eigenschaften, 130
- einfache Anweisungen, 103
- Einfache Typen, 83, 84
- Endlosschleife, 109
- Entwicklerschnittstelle, 175
- Ereignis, 31, 135
- Ereignisbehandlungsroutine, 31, 59
- Ereignisbehandlungsroutinen, 135
- Event Handler, 152
- Event Loop, 152
- Events, 27, 31
- Exception-Hierarchie, 166
- Exceptions, 160
- External-Deklaration, 126
- Extrahieren von Botschaften, 154
- Fehlerbehandlung, 160
- Feld, 129
- Fenster, 52
- Finalization-Teil, 47
- Formular, 24, 52
- For-Schleife, 109
- Forward-Deklaration, 125
- friend, 175
- fundamentale Char-Typen, 101
- fundamentale Integer-Typen, 84
- Fundamentaltypen, 100
- Funktionsdeklaration, 122
- Funktionswert, 119
- generische Integer-Typen, 84, 100
- generische String-Typ, 101
- generische Typen, 100
- generischer Char-Typ, 101
- global, 104, 117
- Globale Bezeichner, 172
- Glyph, 41
- Graphen, 177
- Graphische Formulardatei, 35
- Gültigkeitsbereich, 104, 172
- Gültigkeitsbereichsregeln, 171
- Icons, 36
- IDE, 23
- If-Anweisung, 105
- Implementationsteil, 47
- Implementierung der Methode, 134
- implizite Referenzierung, 177
- indirekte Unit-Referenzen, 49
- Indizierung, 148
- information hiding, 115
- Initialisierungsteil, 47
- Instanz, 128, 129
- integer, 85
- Integer, 84
- Integrated Development Environment, 23
- Integrierter Debugger, 25
- interface, 174
- Interface-Teil, 47
- Kapselung, 127
- Klassen, 128
- Klassenreferenz, 167
- Klassenreferenztypen, 168
- Klassentypen, 118
- Kommentar, 43
- Kompatibilität der Datentypen, 100
- kompilierte Datei, 36
- Komponenten, 29
- Komponentenpalette, 24, 29, 147
- konkateneren, 90
- Konstante, 74
- Konstante Parameter, 116
- Konstruktor, 135
- kurzer String-Typ, 88
- langer String-Typ, 88
- late binding, 128, 140
- Laufvariable, 109
- Laufzeitbibliothek, 161
- Laufzeitfehler, 160
- Liste der aktuellen Parameter, 116
- Liste der formalen Parameter, 116
- Listen, 177
- lokal, 80, 104, 116, 117
- LongBool, 86

- longint, 85
- Mengentyp, 94
- Menü-Designer, 25
- Metaklassen, 168
- Methode, 48, 60, 127, 129
- Microsoft Hilfe-Compiler, 36
- modales Dialogfenster, 69
- Modul, 45
- Nachkomme, 58, 139
- Name, 27
- Namenskonventionen, 16
- nicht typisierte Parameter, 117
- nichtmodales Dialogfenster, 70
- nil, 99
- nullterminierter String, 91
- Objekt, 127, 129, 130
- Objekt-Browser, 25
- Objektinspektor, 24, 26
- objektorientierte Programmierung, 130
- Objektorientierte Programmierung, 127
- Objektreferenz, 167
- Objektselektor, 27
- Objekttyp, 129
- offene Array-Parameter, 117
- Offene Parameter, 178
- offener Array-Parameter, 178
- Online-Hilfe, 20, 23, 27, 31, 61
- OOP, 127, 130
- OOP-Paradigma, 144
- Operatoren, 73
- Ordinale Typen, 84
- owner, 58
- Parameter, 60, 115
- Parameterliste, 115
- parent, 58
- Polymorphie, 128, 140
- PostMessage, 155
- precedence of operators, 73
- private, 175
- Projekt, 33
- Projektdatei, 33, 35, 42
- Projektoptionendatei, 35
- Projektverwaltung, 25, 33
- properties, 130
- Properties, 27, 144
- protected, 175
- Prozedurale Typen, 83
- Prozedurdeklaration, 122
- public, 176
- public, 47
- published, 176
- Qualifizierer, 60
- qualifizierter Bezeichner, 134
- qualifizierter Methodenaufruf, 135
- Quelltexteditor, 24, 30
- Realtyp, 87
- Recordtyp, 95
- Referenz, 177
- Repeat-Schleife, 112
- Reservierte Wörter, 43
- Ressource, 161
- Routine, 31, 72, 114
- Routinenaufruf, 114
- Routinendeklaration, 115
- RTL, 115
- Run Time Library, 115
- Schalter, 29
- Schaltfeld, 29
- Schaltfläche, 29
- Schleifen, 109
- Schnittstelle, 115
- scope, 104
- Scope, 171
- Selektor, 108
- Sender, 60, 138
- shortint, 85
- Sichtbarkeit, 132, 172
- Sichtbarkeitsattribut, 175
- smallint, 85
- Spätes Binden, 128
- Speicheradresse, 77

- Speicherplatz, 77
- static, 177
- Statische Methoden, 128
- String-Typen, 83, 88
- strukturierte Anweisungen, 103
- strukturierte Typen, 92
- Strukturierte Typen, 83
- Subroutine, 114
- Symbolleiste, 24
- Systemsteuerung, 26
- 
- Tabelle virtueller Methoden, 128
- Teilbereichstypen, 86
- Terminator-Nullzeichen, 92
- Titelleiste, 26
- Tokens, 73
- Typ, 76, 82
- Typinformation zur Laufzeit, 166, 176
- Typisierte Konstanten, 79, 176
- 
- unär, 107
- unäre Operatoren, 73
- Unit, 33, 35, 45
- Unit-Kopf, 47
- Unit-Objekt-Code, 36
- uses, 174
- 
- Validierung, 144
- 
- Variable, 76
- Variablenparameter, 116
- Variablenreferenz, 168
- Varianttypen, 83
- Vererbung, 128
- virtual method table, 128
- Virtuelle Methode, 128
- VMT, 128
- Vorfahr, 58, 139
- 
- Wert, 76
- Werteparameter, 116
- While-Schleife, 111
- WideChar, 101
- Windows-API, 149
- Windows-Explorer, 20
- Windows-Hilfdateien, 36
- word, 85
- WordBool, 86
- 
- Zeiger dereferenzieren, 98
- Zeiger referenzieren, 98
- Zeigertyp, 83, 98
- Zirkuläre Unit-Referenzen, 49
- Zusammengesetzte Anweisungen, 103
- Zuweisung, 78
- Zuweisungskompatibilität, 79, 94, 100
- Zuweisungsoperator, 78