

Workshop Visual Basic 6

Detlef Drews, Heinz Schwab

Workshop Visual Basic 6



ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Die Informationen in diesem Produkt werden ohne Rücksicht auf einen
eventuellen Patentschutz veröffentlicht.

Warennamen werden ohne Gewährleistung
der freien Verwendbarkeit benutzt.

Bei der Zusammenstellung von Texten und Abbildungen wurde mit größter
Sorgfalt vorgegangen.

Trotzdem können Fehler nicht vollständig ausgeschlossen werden.

Verlag, Herausgeber und Autoren können für fehlerhafte Angaben
und deren Folgen weder eine juristische Verantwortung noch
irgendeine Haftung übernehmen.

Für Verbesserungsvorschläge und Hinweise auf Fehler sind Verlag und
Herausgeber dankbar.

Alle Rechte vorbehalten, auch die der fotomechanischen Wiedergabe und der
Speicherung in elektronischen Medien.

Die gewerbliche Nutzung der in diesem Produkt gezeigten Modelle und Arbeiten
ist nicht zulässig.

Fast alle Hardware- und Softwarebezeichnungen, die in diesem Buch erwähnt werden,
sind gleichzeitig auch eingetragene Warenzeichen oder sollten als solche betrachtet werden.

Umwelthinweis:

Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.

Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem
und recyclingfähigem PE-Material.

10 9 8 7 6 5 4 3 2 1

03 02 01 00

ISBN 3-8273-1664-2

© 2000 by Addison-Wesley Verlag,
ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Alle Rechte vorbehalten

Einbandgestaltung: Rita Fuhrmann, Frankfurt/Oder

Lektorat: Christina Gibbs, cgibbs@pearson.de,

Koorektorat: Simone Burst, Großberghofen

Herstellung: TYPisch Müller, Arcevia, Italien

Satz: reemers publishing services gmbh, Krefeld

Druck: Media-Print, Paderborn

Printed in Germany

Inhaltsverzeichnis

Vorwort	11
1 Einleitung	13
1.1 Visual Basic 6.0	13
1.2 Ein Programmentwicklungssystem für alle	13
1.3 Die Leistungsmerkmale der Editionen	16
2 Workshop: Die Entwicklungsumgebung	19
2.1 Die Oberfläche von Visual Basic 6.0	19
2.2 Die wichtigsten Menüs	20
2.2.1 Menü Datei	20
2.2.2 Übung: Anlegen eines neuen Projekts	21
2.2.3 Menü Bearbeiten	22
2.2.4 Menü Ansicht	23
2.2.5 Menü Projekt	23
2.2.6 Menü Format	25
2.2.7 Menü Debuggen	25
2.2.8 Übung: Programm Debuggen	26
2.2.9 Menü Ausführen	29
2.2.10 Menü Extras	29
2.2.11 Menü Add-Ins	30
2.3 Symbolleisten	30
2.4 Das Eigenschaftfenster	30
2.5 Der Projekt-Explorer	32
2.5.1 Kontextmenüs des Projekt-Explorers	32
2.6 Das Code-Fenster	34
2.7 Die Werkzeugsammlung und das Formularfenster	35
2.7.1 Die Werkzeugsammlung – Objekte hinzufügen ...	36
3 Workshop: Variablen und Konstanten	39
3.1 Variablen	39
3.1.1 Die Deklaration von Variablen	39
3.1.2 Die implizite Deklaration	40
3.1.3 Die explizite Deklaration	40
3.1.4 Syntax einer Variablendeklaration	42

3.2	Datentypen und Wertebereiche	43
3.2.1	Übung: Deklaration von Variablen	44
3.3	Werte in Variablen speichern	48
3.3.1	Einen Wert im Datentyp Date speichern	49
3.4	Zweimal Datentyp String.	49
3.4.1	Übung: Dynamische und statische String	52
3.5	Der multifunktionale Datentyp Variant im Detail	54
3.5.1	VarType	54
3.5.2	Übung: Verwendung des Datentyps Variant	55
3.6	Benutzerdefinierte Datentypen	58
3.6.1	Übung: Verwendung benutzerdefinierter Datentypen	61
3.7	Feldvariablen	63
3.7.1	Übung: Programmieren mit einer Feldvariablen	64
3.7.2	Unter- und Obergrenze einer Feldvariablen ermitteln	66
3.7.3	Übung: Grenzen von Feldern ermitteln	66
3.8	Konstanten	67
3.8.1	Literale	67
3.8.2	Symbolische Konstanten	67
3.8.3	Vordefinierte Konstanten	68
3.9	Gültigkeitsbereiche von Variablen und Konstanten	69
3.9.1	Gültigkeitsbereich von Variablen	70
3.10	Programmieren mit Variablen	72
3.10.1	Klassifizierung von Operatoren	72
3.10.2	Die Rechenoperatoren	72
3.10.3	Übung: Berechnung eines Rabatts	75
3.10.4	Die Vergleichsoperatoren	77
3.10.5	Übung: Verwendung von Vergleichsoperatoren	79
3.10.6	Die logischen Operatoren	80
3.10.7	Übung: Verwendung von logischen Operatoren	83
4	Workshop: Steuerung des Programmablaufs	87
4.1	Entscheidungsstrukturen.	87
4.1.1	Die If...Then-Anweisung	87
4.1.2	Übung: Verwendung der If...Then-Struktur	88
4.1.3	Übung: Verhindern der Division durch 0 mit If...Then	92
4.1.4	Die Select-Case-Struktur	95
4.1.5	Übung: Verwenden der Select...Case-Struktur	96

4.1.6	Die GoTo-Anweisung	98
4.1.7	Fehlerbehandlung: On Error GoTo	99
4.1.8	Übung: Schreiben einer Fehlerbehandlungsroutine	100
4.2	Programmschleifen	107
4.2.1	Die For...Next-Schleife	107
4.2.2	Übung: Verwenden der For...Next-Anweisung . . .	108
4.2.3	Übung: Berechnung der Fakultät	112
4.2.4	Übung: Ausgabe einer mehrzeiligen Tabelle	114
4.2.5	Die Do...Loop-Schleife	117
4.2.6	Übung: Verwenden der Do...Loop-Schleife	119
4.2.7	Übung: Zeilenweises Einlesen einer Datei	123
4.2.8	Übung: Eine Warteschleife	128
5	Workshop: Prozeduren und Funktionen	133
5.1	Der Benutzerdialog MsgBox	135
5.1.1	Übung: Formatierte Ausgabe mit der Funktion MsgBox	142
5.1.2	Übung: Dialog mit Rückgabe und Auswertung . . .	143
5.2	Der Benutzerdialog InputBox	146
5.2.1	Übung: Verwenden der InputBox-Funktion	149
5.3	Funktionen zur Verarbeitung von Zeichenketten	153
5.3.1	Zeichenketten aneinander hängen	153
5.3.2	Länge einer Zeichenkette ermitteln	154
5.3.3	Entfernen von vor- oder nachlaufenden Leerstellen	155
5.3.4	Umwandeln von Zeichenketten in Groß- oder Kleinbuchstaben	156
5.3.5	Ausschneiden von Teilstrings	157
5.3.6	Übung: Ausschneiden aus einer Zeichenkette	159
5.3.7	Ersetzen von Teilstrings	163
5.3.8	Suchen von Teilstrings in Strings	164
5.3.9	Übung: Suchen und Ersetzen von Zeichen	166
5.3.10	Übung: Textdrehen	169
5.3.11	Umwandlung einer Zeichenkette in eine Zahl	171
5.3.12	Umwandlung einer Zahl in eine Zeichenkette	172
5.3.13	Formatieren von Strings	173
5.3.14	Übung: Formatierte Ausgabe von Zahlen	175
5.3.15	Übung: Uhr	180
5.4	Rechenfunktionen	182
5.4.1	Übung: Lottozahlen Quicktipp	183
5.5	Umwandlungsfunktionen	185

5.6	Programmieren von Prozeduren und Funktionen	186
5.6.1	Prozeduren	187
5.6.2	Übung: Prozedur Textumdrehen	190
5.6.3	Übung: Positioniere ein Formular	193
5.6.4	Funktionen	199
5.6.5	Übung: Funktion Textumdrehen	200
6	Workshop: Mit den Steuerelementen programmieren.	203
6.1	Arbeiten mit der Werkzeugsammlung.	203
6.1.1	Übung: Zusätzliche Objekte und Steuerelemente einfügen	204
6.2	Mit Steuerelementen im Formular arbeiten	206
6.2.1	Übung: Richten Sie Steuerelemente aus	208
6.3	Ereignisgesteuert programmieren	213
6.3.1	Ein Steuerelement im Programm ansprechen	214
6.4	Das Steuerelement »Form (Formular)«	216
6.4.1	Übung: Formulare laden und anzeigen	217
6.4.2	Übung: Unterschiedliche Menütypen im Formular integrieren	220
6.4.3	Übung: Im Formular Freihand zeichnen	227
6.4.4	Übung: Linien gezielt im Formular zeichnen	229
6.4.5	Übung: Formular bildschirmmittig positionieren	230
6.4.6	Übung: Formular wie bei letzter Benutzung positionieren	232
6.5	Das Steuerelement »CommandButton«	235
6.5.1	Übung: Auf Anforderung einen Button ein- oder ausblenden	235
6.5.2	Übung: Button mit Grafik und QuickInfo	237
6.5.3	Übung: Der Button hüpfert weg!	238
6.5.4	Übung: Buttongröße dynamisch zur Formgröße anpassen	239
6.5.5	Übung: Individuelle Programmschnellstartleiste (Application Caller)	241
6.6	Das Steuerelement »Frame«	250
6.6.1	Übung: Auf Anforderung einen Rahmenstil, Rahmentext setzen	250
6.7	Das Steuerelement »DriveListBox«	252
6.7.1	Übung: Einen Laufwerkswechsel anzeigen	252
6.8	Das Steuerelement »DirListBox«	253
6.8.1	Übung: Ordner eines Laufwerks listen	253
6.9	Das Steuerelement »FileListBox«	255
6.9.1	Übung: Dateien eines Ordners listen	255

6.10	Das Steuerelement »Label«	257
6.10.1	Übung: Text laden und zentrieren	257
6.10.2	Übung: Benutzeroberfläche zur Laufzeit anpassen	259
6.10.3	Übung: Entwicklung eines Form-Ausgabeinterpreters	262
6.11	Das Steuerelement »TextBox«	275
6.11.1	Übung: Eingabe von Passwörtern	275
6.11.2	Übung: Numerische Eingabeprüfung	278
6.11.3	Übung: Ein kleiner Editor gefällig	279
6.11.4	Übung: Entwicklung eines Form-Eingabeinterpreters	281
6.12	Das Steuerelement »OptionButton«	290
6.12.1	Übung: Farben einstellen	291
6.13	Das Steuerelement »CheckBox«	294
6.13.1	Übung: Text formatieren	295
6.14	Das Steuerelement »ComboBox«	296
6.14.1	Übung: Aus einer Artikelliste auswählen	297
6.15	Das Steuerelement »ListBox«	299
6.15.1	Übung: Musiktitel einer ListBox hinzufügen	300
6.15.2	Übung: Musiktitel aus der ListBox auslesen	301
6.15.3	Übung: Musiktitel aus der ListBox löschen	302
6.15.4	Übung: Alle Musiktitel aus der ListBox löschen	303
6.15.5	Übung: Musiktitel sortieren und multiselektierbar machen	304
6.16	Das Steuerelement »ScrollBars«	305
6.16.1	Übung: DM-Betrag über horizontale Bildlaufleiste wählen	305
6.17	Das Steuerelement »PictureBox«	307
6.17.1	Übung: Auf Anforderung ein Bild laden und anzeigen	307
6.17.2	Übung: Bild im Formular zur Laufzeit frei bewegen	308
6.18	Das Steuerelement »Timer«	309
6.18.1	Übung: Uhrzeit anzeigen	309
6.18.2	Übung: Entwicklung eines Terminplaners	310
6.18.3	Übung: Stoppuhr – Zeiten stoppen und monetär bewerten	317
6.19	Das Steuerelement »Image«	325
6.19.1	Übung: Auf Anforderung ein Bild laden, anzeigen und dessen Größe anpassen	325
6.19.2	Übung: Ein Bildbetrachter (PictureBox) gefällig	327

6.20	Das Steuerelement »Data«	339
6.20.1	Übung: Durch den Datenbestand einer Access-Datenbank blättern	339
6.21	Das Steuerelement »DBList«	342
6.21.1	Übung: Den Datenbestand einer Access-Datenbank listen	342
	Stichwortverzeichnis	345

Vorwort

Gestatten, dass wir uns Ihnen kurz vorstellen:

Mein Name ist Detlef Drews, Wirtschaftsinformatiker. Ich bin seit 1993 als selbstständiger Unternehmer erfolgreich in den Bereichen IT-Beratung, Software-Engineering, EDV-Schulung und -Publikation tätig und erarbeite primär für mittelständische Unternehmen mit heterogener EDV-Infrastruktur Client/Server- und PPS-Softwarelösungen.

Und mein Name ist Heinz Schwab, Dipl.-Ing. (FH). Ich bin seit 1990 als Produktmanager und Softwareentwickler für Kommunikationslösungen im Netzwerkbereich tätig. Auch die Beratung und Schulung von Kunden im In- und Ausland gehört zu meinem Aufgabengebiet.

Zudem arbeiten wir als Team schon seit vielen Jahren zusammen und haben uns so als Fachautoren, Buchautoren und Herausgeber zahlreicher EDV-Fachpublikationen, -Bücher und -Nachschlagewerke zusätzlich einen Namen gemacht.

Visual Basic ist heute weltweit das am meisten eingesetzte Entwicklungssystem für Windows, das Leichtigkeit beim Erlernen, Leistungsfähigkeit und flexible Erweiterbarkeit auf einzigartige Weise kombiniert.

Und so richtet sich dieses Buch an alle, die sich schon mit einer bestimmten Thematik in Visual Basic (u.a. z.B. mit Büchern aus der Lernen- oder Go To-Reihe etc.) auseinander gesetzt haben und nun ihr Wissen noch weiter vertiefen wollen.

In Workshops zu den gängigen Programmierthemen vertiefen und erweitern Sie durch praxisorientierte, unterschiedlich schwere Übungen Ihren Visual Basic Kenntnisstand und bilden sich so sukzessive zum Visual Basic-Softwareentwickler weiter.

Alle im Arbeitsbuch behandelten workshopspezifischen Übungen befinden sich als gelöste Visual Basic-Applikation auf der beiliegenden CD in dem Ordner des jeweils gleichlautenden Buchkapitels. Der mitgelieferte Quellcode wurde mit Visual Basic 6.0 Professional Edition unter dem Betriebssystem Windows95/98 erstellt und eingesetzt.

Bleibt uns wohl nur noch, Ihnen viel Freude und Erfolg mit diesem Arbeitsbuch zu wünschen. Falls Sie Fragen, Anregungen oder Kritiken haben, so können Sie sich über die angegebenen Adressen auch gerne direkt an uns wenden.

Wer sind WIR?

Für wen ist dieses Buch konzipiert?

Es ist ein praktisches Arbeitsbuch

Zu guter Letzt möchten wir uns noch ganz herzlich bei unseren verständnisvollen Familien bedanken, die durch Ihre Unterstützung den notwendigen Freiraum für dieses Projekt schafften – danke.

Villingen-Schwenningen im September 2000

Detlef Drews;
Heinz Schwab;

DetlefDrews@t-online.de
HeinzSchwab@swol.de

Verwendete Symbole

Folgende Symbole werden verwendet:



Beispiele helfen Ihnen, Ihre Kenntnisse in der Visual Basic-Programmierung zu vertiefen. Sie werden mit diesem Icon gekennzeichnet.



Hinweise machen auf Elemente aufmerksam, die nicht selbstverständlich sind.



Achtung, mit diesem Icon wird eine **Warnung/Fehlerquelle** angezeigt. An der markierten Stelle sollten Sie aufmerksam sein.



Manches geht ganz leicht, wenn man nur weiß, wie. **Tipps & Tricks** finden Sie in den Abschnitten, wo dieses Icon steht.

Jede **Übung** besitzt eine Bewertung der Schwierigkeit. Dabei gelten im Allgemeinen folgende Richtlinien:



Die Übung besteht aus einfachen elementaren Konstrukten, die vielleicht sogar in ähnlicher Form bereits als Beispiel vorgetragen wurden.



Die Übung verlangt die Kombination mehrerer Gebiete der Sprache. Dies können zum Beispiel Themen sein, die bereits in vorhergehenden Kapiteln besprochen und geübt wurden.



Die bisher besprochenen Elemente der Sprache müssen kombiniert und mit ihnen ein Problem gelöst werden, welches ein gewisses Maß an Transferleistung erfordert. Das eventuell benötigte themenfremde Wissen zur Lösung der Übung wird in der Aufgabenstellung vermittelt.

1 Einleitung

1.1 Visual Basic 6.0

Visual Basic (VB) gibt es als Programmentwicklungssystem schon seit vielen Jahren. Es begann mit der Version 1.0 für das damals aktuelle Windows 3.0. Mit den nachfolgenden Versionen von Windows hat sich auch Visual Basic stetig verbessert und an die neuen Eigenschaften und Möglichkeiten angepasst. Visual Basic 6.0 ist nun die neueste Version dieser äußerst erfolgreichen Programmiersprache.

32-Bit-Umgebung

Sie ist voll auf die 32-Bit-Umgebung von Windows95 und Windows NT zugeschnitten und unterscheidet sich damit deutlich von den 16-Bit-Versionen für Windows 3.x.

War es mit Visual Basic 4.0 noch möglich, Programme für 16-Bit- und 32-Bit-Windows-Betriebssysteme zu erstellen, so steht seit der Version 5.0 ein Programmentwicklungssystem nur für 32-Bit-Applikationen zur Verfügung.



Visual Basic 6.0 bietet gegenüber den Vorgängerversionen eine Vielzahl von neuen Möglichkeiten und Leistungsverbesserungen. Hierzu zählen ActiveX und damit verbunden die Internetprogrammierung. Auch im Bereich der Programmerstellung und Fehlersuche gibt es hilfreiche Neuerungen, welche die Arbeit erleichtern und neue Möglichkeiten erschließen.

1.2 Ein Programmentwicklungssystem für alle

Mit Visual Basic war es erstmals möglich, Programme für die komplexe grafische Benutzeroberfläche von Windows auf einfachste Art zu erstellen; und das mit Hilfe einer weit verbreiteten und einfachen Programmiersprache wie Basic.

Trotz der immer weiter gestiegenen Funktionalität des Betriebssystems, des großen Leistungsumfangs und der sich daraus ergebenden Komplexität hat sich Visual Basic den Ruf als einfach zu erlernendes Programmentwicklungssystem erhalten können.

Es ist immer noch auf einfachste Weise möglich, ein Programm zu erstellen, das die Anforderungen einer grafischen Benutzeroberfläche erfüllt. Daneben gibt es allerdings auch viele mächtige Funktionen und Hilfsmittel, die es ermöglichen, den Leistungsumfang des Betriebssystems voll auszuschöpfen.

Dies spiegelt sich auch in den verschiedenen Editionen von Visual Basic wider.



Seit der Version 4.0 liegt das Programmiersystem in einer Standard, einer Professional und einer Enterprise Edition vor. Visual Basic 6.0 ist ebenfalls in diesen drei Stufen erhältlich. Die Bezeichnungen sagen bereits einiges über den Leistungsumfang der verschiedenen Editionen aus.

Standard Edition

Die Standard Edition beinhaltet lediglich die Basisfunktionalität, die ein professionelles und komplexes Arbeiten nur sehr eingeschränkt erlaubt. Es sind nur die wichtigen Steuerelemente im Lieferumfang enthalten, wie etwa das Grid-Steuerelement für Tabellen und datengebundene Steuerelemente.

Dem Programmieranfänger wird sicherlich die Standard Edition von Visual Basic genügen. Hier sind alle Elemente vorhanden, um mit wenigen Schritten eine funktionale, grafische Benutzeroberfläche zu erstellen. Dabei müssen Sie sich nicht mit den besonderen Restriktionen der Betriebssystemumgebung beschäftigen, denn Visual Basic nimmt Ihnen diese Arbeit ab.

Mit wenigen Mausklicks kann das grafische Gerüst der Oberfläche aufgebaut werden, die dann durch einfach zu handhabenden Basiccode mit Leben gefüllt wird.



Lediglich als Benutzer anderer Programmiersprachen, sprich Umsteiger, werden Sie sich auf die in Visual Basic verwendete ereignisorientierte Programmierung umstellen müssen. Es gibt bei dieser Technik kein Hauptprogramm mehr. Die einzelnen Prozesse werden beim Auftreten von bestimmten Ereignissen angestoßen.

Professional Edition

Die Professional Edition bietet dagegen für den erfahrenen Programmierer alles, was die Arbeit mit Visual Basic beschleunigt und vor allen Dingen komfortabler gestaltet. Als Erweiterung zu den Grundfunktionen der Standard Edition enthält die Professional Edition eine ganze Reihe zusätzlicher und zum Teil neuer Steuerelemente. Des Weiteren kommen jede Menge Tools und Funktionen hinzu, deren Nutzung in der Standard-(Learning-)Edition nicht möglich ist oder die erst gar nicht vorhanden sind.

Enterprise Edition

Die Enterprise Edition schließlich ist, wie ihr Name schon sagt, vor allem für Unternehmen geeignet, in denen größere Projektteams in verteilten Umgebungen an einem Projekt entwickeln (Client-Server). Sie enthält neben allen Werkzeugen der Professional Edition, auch neue zusätzliche Möglichkeiten zur Versionskontrolle und Leistungsanalyse sowie etliche verbesserte Werkzeuge für die Datenbankbearbeitung etc.

Die Entwicklungsumgebung

Die Entwicklungsumgebung von Visual Basic bietet Ihnen einen gut strukturierten Überblick über die möglichen Ereignisse bereits im Editor, so dass der Code zum entsprechenden Ereignis problemlos eingegeben werden kann.

Beim Schreiben des Programmcodes unterstützt Visual Basic Sie durch eine Syntaxprüfung bereits bei der Eingabe. Zusätzlich führt Visual Basic beim Start eines Programms in der Entwicklungsumgebung noch einen Test des Gesamtprogramms durch und zeigt die gefundenen Fehler an.

Erst nach bestandener Prüfung wird der Interpreter von Visual Basic gestartet und der Programmcode abgearbeitet. Sollte nun während des Programmlaufs ein Fehler auftreten, so stoppt Visual Basic die Ausführung und gibt Ihnen die Möglichkeit, die fehlerhafte Zeile direkt zu korrigieren.

Anschließend kann der normale Programmablauf mit der korrigierten Zeile wieder aufgenommen werden. Sie können Ihr Programm nun so lange testen und verändern, bis es das gewünschte Ergebnis liefert. Ist alles fehlerfrei lauffähig, können Sie sich aus Ihrem Programmcode ein ausführbares Programm generieren lassen.

Steuerelemente

Sollten die mitgelieferten Steuerelemente nicht mehr ausreichen, um bestimmte funktionelle Anforderungen zu erfüllen, so ist es immer möglich, spezielle Steuerelemente (Komponenten) zusätzlich zu erwerben und dann einzubinden.

Der erfahrene Programmierer

Als bereits erfahrener Programmierer, dessen Anforderungen nicht mehr mit der Einsteiger-Edition zu befriedigen sind, ist für Sie sicher die Professional Edition die richtige Programmierumgebung.

Hier bieten sich allein durch die zusätzlichen Steuerelemente wesentlich mehr Möglichkeiten. Sollten die Steuerelemente jedoch nicht Ihren Anforderungen entsprechen, sind Sie sogar in der Lage, eigene Steuerelemente zu erstellen.

Diese können in späteren Programmen wieder benutzt werden und es lässt sich so wertvolle Zeit einsparen. Hierbei helfen auch die zahlreichen Add-In-Tools, die alle von Assistenten unterstützt werden.

Diese wiederum können Ihnen viel Arbeit abnehmen. Wollen Sie intensiv mit Datenbanken arbeiten, so kommen Sie am Einsatz der Professional oder gar der Enterprise Edition nicht vorbei.

Die Enterprise Edition gibt dem Entwickler eine Programmierumgebung mit eingebauten Werkzeugen zum einfachen Erstellen, Testen und Anwenden von verteilten Client-Server-Großanwendungen an die Hand. Diese Edition ist für die Entwicklung von Unternehmenssoftware im Team prädestiniert.

1.3 Die Leistungsmerkmale der Editionen

Die Version 6.0 von Visual Basic stellt eine konsequente Weiterentwicklung der bisherigen Versionen in Richtung Objektorientierung dar. Es war zwar in den ersten Versionen bereits möglich, objektorientiert zu programmieren; es gab jedoch keine standardisierten Mechanismen, auch nicht vonseiten des Betriebssystems, die eine solche Entwicklung konsequent unterstützt hätten.



Erst mit der Einführung von *COM*, *DCOM* und dessen Ablegern im Zuge von Windows95 und WindowsNT konnten die Entwicklungssysteme diese Technik mit Assistenten und Steuerelementen dem Programmierer näher bringen.

Visual Basic 6.0 unterstützt im Rahmen seiner Möglichkeiten diese objektorientierten Architekturen und Methoden und ermöglicht es Ihnen demnach, sich den Anforderungen moderner Softwareentwicklung zu stellen.

Standard (Learning) Edition

- ▶ Es ist möglich, mehrere Projekte gleichzeitig zu bearbeiten, so dass Sie beispielsweise zur Fehlersuche Ihr Testprojekt und etwa ein ActiveX-Steuerelementprojekt gemeinsam ausführen können.
- ▶ Das Erscheinungsbild der Entwicklungsumgebung ist defaultmäßig den Visual-Studios angepasst. Es ist also eine Multiple-Document-Interface-Oberfläche(MDI).
Auf Wunsch lässt sich jedoch die von den bisherigen Versionen bekannte Oberfläche mit Single-Document-Interface aktivieren.
- ▶ Es können eigene Ereignisse in Klassenmodulen definiert und in anderen Objekten empfangen und ausgewertet werden.
- ▶ Der *Editor* erzeugt automatisch Syntaxinformationen zu einer Anweisung/Funktion (QuickInfo), stellt automatisch den Code während des Editierens fertig, erlaubt eine blockweise Kommentarverwaltung, die Anzeige von Eigenschaften zu einem Objekt und bietet Drag&Drop-Möglichkeiten.
- ▶ *Polymorphismus*, bekannt aus der objektorientierten Sprachwelt, ist ebenfalls in Visual Basic-Klassen möglich.
- ▶ Steuerelemente zur Anzeige von Bildern unterstützen die Formate *.gif* und *.jpg*, welche in Internetseiten häufig verwendet werden.
- ▶ Alle Steuerelemente besitzen eine *ToolTip*-Eigenschaft, die am Mauszeiger eine Kurzinfo zum Objekt erscheinen lässt.
- ▶ Eigene Klassenmodule können einem Debugging unterzogen werden.
- ▶ Eigenschaften können mit optionalen Argumenten erstellt werden.

Professional Edition

Sie enthält alle Eigenschaften und Möglichkeiten der Standard Edition erweitert u.a. um folgende:

- ▶ Programme können kompiliert werden. Dabei werden diese in den so genannten »systemeigenen Code« übersetzt, der eine spürbar höhere Abarbeitungsgeschwindigkeit zulässt. Die Art der Kompilierung lässt sich durch Optionen optimieren (Größe, Geschwindigkeit etc.).
- ▶ Sie können mit der Entwicklungsumgebung eigene ActiveX-Steuerelemente entwerfen. Hierbei unterstützt Visual Basic den ActiveX-Standard von Microsoft, der auf dem Component-Object-Model basiert und eine Weiterentwicklung zu OLE (Object Linking and Embedding) darstellt.

Dies ermöglicht nicht nur die Entwicklung von objektorientierten Programmen und Komponenten, die auch außerhalb von Visual Basic wiederverwendet werden können, sondern auch auf einfachste Weise die Erstellung von ActiveX-Dokumenten, die im Internet Anwendung finden.

- ▶ Es gibt *Assistenten* u.a. für ActiveX-Steuerelementprojekte, für ActiveX-Dokumente, für Datenformulare und für ActiveX-Eigenschaftenseiten.
- ▶ Ein *Komponentendownload* ist gewährleistet und ermöglicht dem Benutzer eines Programms/Internet-Dokuments, die von Ihnen erstellten ActiveX-Steuerelemente aus dem Internet in seinen Browser zu laden.
- ▶ HTML-Dokumente können erstellt und automatisch im Microsoft Internet Explorer angezeigt werden.
- ▶ Sie können mit einem speziell dafür geschaffenen Designer *DHTML*-Seiten gestalten.
- ▶ Einheitlicher Datenzugriff über das neue *ADO (ActiveX Data Object)*-Steuerelement.

Enterprise Edition

Sie enthält neben den bisher genannten Funktionen u.a.:

- ▶ Prozessinterne Komponenten wie DLLs, die keiner Interaktion mit dem Benutzer bedürfen, können für den Einsatz in »multithreading«-Applikationen erstellt werden (*Multithreading Dlls*).
- ▶ Die Verteilung von Komponenten mittels *Distributed COM* wird vollständig unterstützt.
- ▶ Ereignisgesteuerte Datenbankoperationen ermöglichen es, asynchrone Operationen zu implementieren. Dadurch entfallen periodische Abfragen auf Ergebnisse hin.

1 Einleitung

- ▶ Mit dem *Verbindungs-Designer* können Sie zur Entwurfszeit Verbindungs- und Abfrageobjekte zum Definieren von Eigenschaften und Methoden vorab erstellen.
- ▶ Integriertes *Microsoft Repository*, bei dem es sich um ein Hilfsmittel handelt, das vor allem bei der Teamentwicklung eingesetzt werden kann. Es beinhaltet eine Verwaltung des Gesamtprojekts bezüglich den einzelnen, gemeinsam genutzten und entwickelten Komponenten eines Software-Systems.
- ▶ Ein Wizard Manager erlaubt es Ihnen, einen eigenen Assistenten zu erstellen, den Sie mit in Ihr Programm aufnehmen können.
Weitere Hilfsmittel und Add-Ins sind u. a.:
- ▶ Zum Testen von RemoteAutomation: der *ApplicationPerformance Explorer*.
- ▶ Zur Projekt- und Versionsverwaltung auch für Teams: der *VisualSourceSafe*.

2

Workshop: Die Entwicklungsumgebung

Visual Basic ist zum einen eine Programmiersprache, zum anderen jedoch auch ein hochintegriertes Werkzeug, um Programme zu erstellen.

Sie erhalten Visual Basic immer als Komplettpaket, bestehend aus der Programmiersprache selbst, den notwendigen Werkzeugen, die Sie bei der Erstellung eigener Programme unterstützen, und einer Anzahl von Zusatzprogrammen, die oft benötigte Aufgaben stark vereinfachen. All dies ist bei Visual Basic in der so genannten Entwicklungsumgebung zusammengefasst.

In diesem Kapitel wird die Entwicklungsumgebung in aller Kürze vorgestellt. Die wichtigsten Oberflächenelemente und Werkzeuge werden benannt und gezeigt. Zudem wird besprochen, was ein Visual Basic-Projekt ist.

Programmiersprache und Entwicklungsumgebung

2.1 Die Oberfläche von Visual Basic 6.0

Wenn Sie Visual Basic 6.0 starten, sehen Sie das Fenster aus Abbildung 2.1. Falls Sie Visual Basic zum ersten Mal gestartet haben, werden Sie feststellen, dass eine gewisse Einarbeitungszeit benötigt wird, um alle Elemente der Oberfläche kennen zu lernen und mit ihnen arbeiten zu können.

Oberfläche

Aber denken Sie daran, diese Entwicklungsumgebung wird Ihnen die Entwicklung Ihrer Programme auf einfachste Weise erlauben, Sie müssen sie eben nur erst kennen lernen.

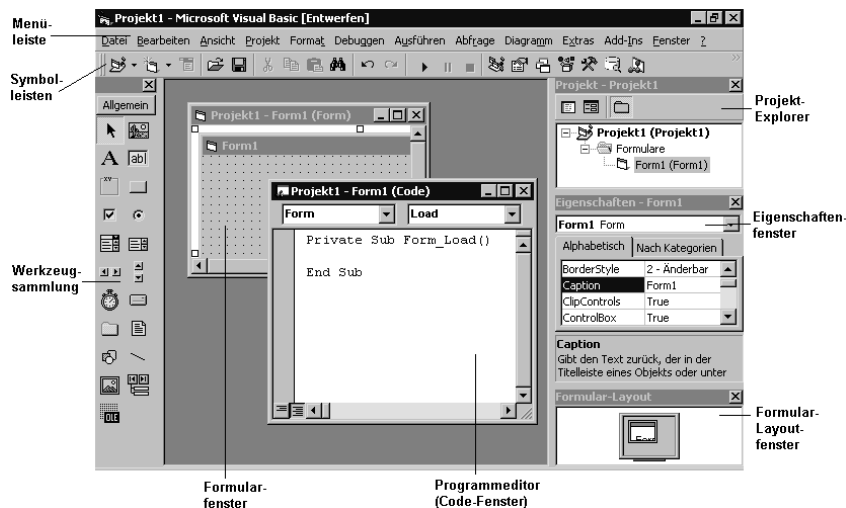


Abbildung 2.1: Die Visual Basic 6.0 Entwicklungsumgebung

Um Ihnen einen leichten Einstieg zu ermöglichen, werden in den folgenden Abschnitten zunächst die wichtigsten Elemente der Oberfläche benannt und erläutert.

2.2 Die wichtigsten Menüs

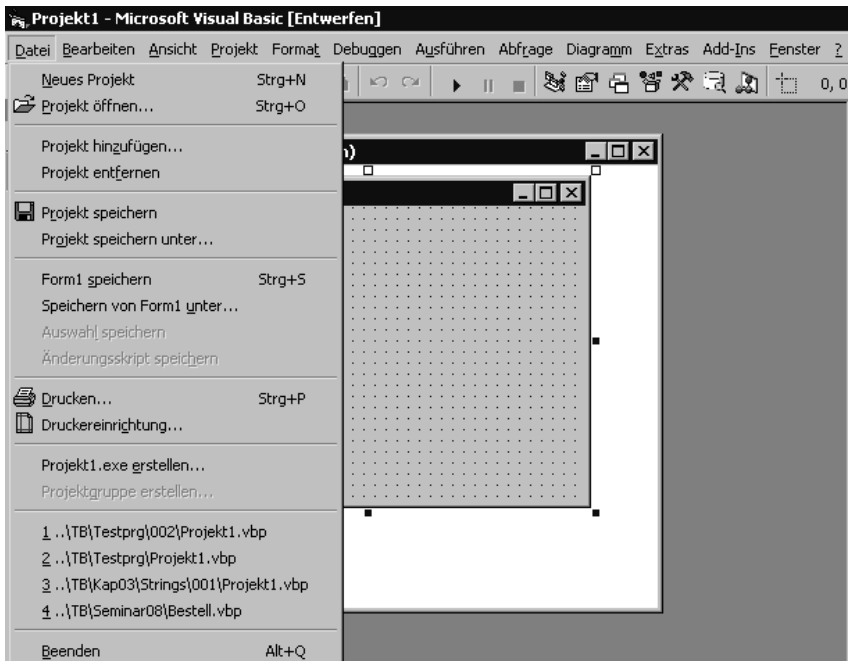
Für einen ersten Überblick werden in den folgenden Abschnitten die Menüs von Visual Basic gezeigt und kurz erläutert. Viele Funktionen der Entwicklungsumgebung können nicht nur über die *Menüs*, sondern auch über *HotKeys*, *Funktionstasten* oder objektbezogene *PopUp-Menüs* aufgerufen werden.

2.2.1 Menü Datei

Menü Datei Im Menü DATEI (Abbildung 2.2) finden Sie alle Funktionen, die notwendig sind, um ein Projekt oder einzelne Dateien des Projektes zu erstellen, zu laden oder zu speichern.

Sie können in diesem Menü Ausdrücke starten, das Programm erstellen (kompilieren) und Visual Basic beenden. Darüber hinaus wird in diesem Menü eine Liste der zuletzt geöffneten Projekte geführt, die dazu verwendet werden kann, diese Projekte zu laden, ohne dafür einen entsprechenden Dialog öffnen zu müssen.

Abbildung 2.2:
Das Menü Datei



2.2.2 Übung: Anlegen eines neuen Projekts

Legen Sie ein Standardprojekt an und speichern Sie es.

Da dieses Projekt keinen Programmcode hat, ist es auf der beiliegenden CD nicht gespeichert.

Lösung

Über das Menü DATEI->NEUES PROJEKT wird der Dialog NEUES PROJEKT (Abbildung 2.3) geöffnet.

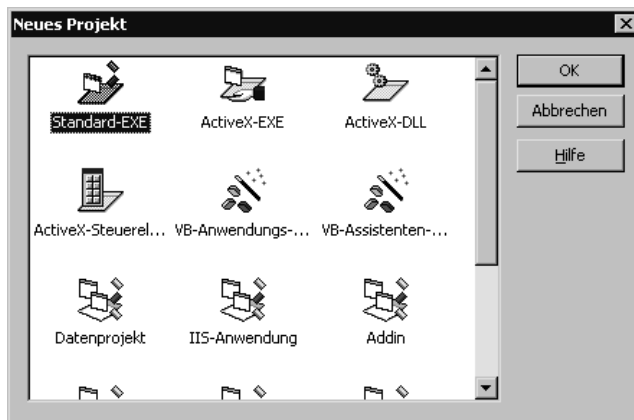


Abbildung 2.3:
Dialog Neues
Projekt

Wählen Sie den Eintrag *Standard-EXE* und betätigen Sie die Schaltfläche *OK*. Visual Basic erstellt jetzt automatisch ein Projekt mit allen notwendigen Projektdateien. Das zugehörige Modul wird automatisch geöffnet, so dass Sie mit dem Oberflächendesign sofort anfangen können.

Standard-EXE

Bei einer *Standard-EXE* wird dem Projekt ein Formular hinzugefügt, wie Sie im Projekt-Explorer (Abbildung 2.4) erkennen können.

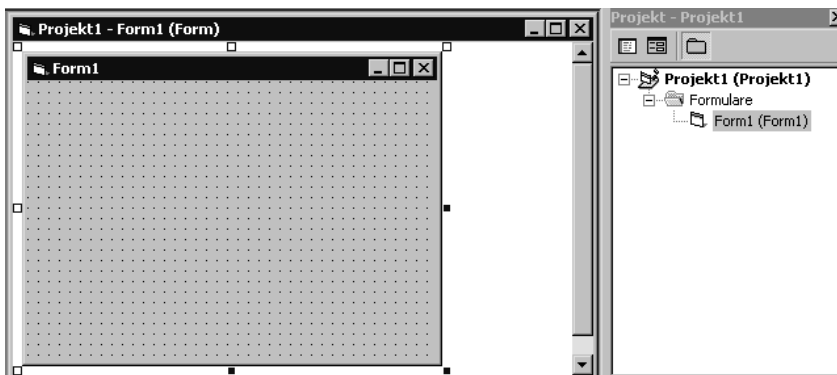


Abbildung 2.4:
Ein Standard-
Projekt wurde
angelegt

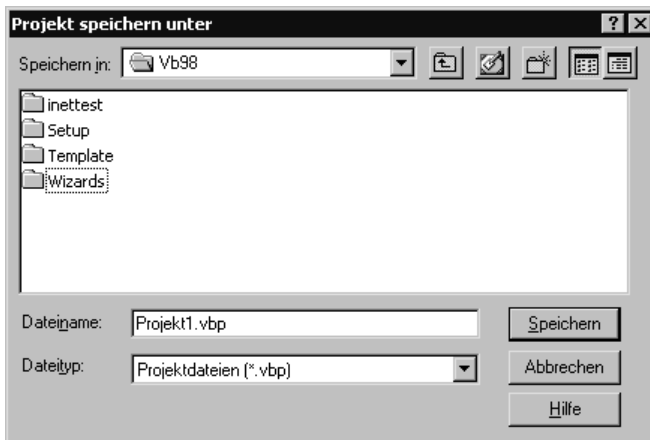
Projekt speichern Um das Projekt zu speichern, ist wiederum eine Auswahl des Menüs DATEI erforderlich. Die einfachste Art ist die Auswahl des Menüpunktes PROJEKT SPEICHERN. Dieser sorgt dafür, dass alle dem Projekt zugehörigen Dateien gespeichert werden.

Wurde das Projekt bisher nicht gespeichert, so erhalten Sie Dialoge, die Ihnen die Möglichkeit geben jeder einzelnen Projektdatei einen Namen zu geben und das Verzeichnis auszuwählen, in dem sie gespeichert werden soll.

das Formular speichern Bei unserem Projekt werden nacheinander zwei Dialoge aufgeblendet. Der erste Dialog verlangt die Speicherung der Datei *Form1.frm*. Mit der Endung *frm* werden Formulare gespeichert. Es handelt sich um das Formular *Form1*, das gespeichert werden soll.

die Projektdatei speichern Ein zweiter Dialog (Abbildung 2.5) verlangt das Speichern der Datei *Projekt1.vbp*. Mit dieser Endung wird das Projekt selbst gespeichert. In der Projektdatei sind alle Informationen enthalten, die notwendig sind um das Projekt zu einem späteren Zeitpunkt wieder zu laden.

Abbildung 2.5:
Speichern der
Projektdatei



Verwenden Sie beim Speichern von Projekten sprechende Namen. Es ist sonst mühsam, unter durchnummerierten Dateien eine bestimmte zu finden.

2.2.3 Menü Bearbeiten

Menü Bearbeiten Das Menü BEARBEITEN (Abbildung 2.6) bietet neben den nun fast zum Standard gewordenen Punkten wie *Rückgängig*, *Kopieren*, *Ausschneiden*, *Einfügen*, *Suchen* und *Ersetzen* etc. einige neue und hilfreiche Funktionen.

Diese betreffen vor allem den Editor und dienen der Fehlerminimierung bei der Eingabe von Code. Eine nähere Beschreibung dieser Hilfsmittel finden Sie im Abschnitt über den *Editor*.

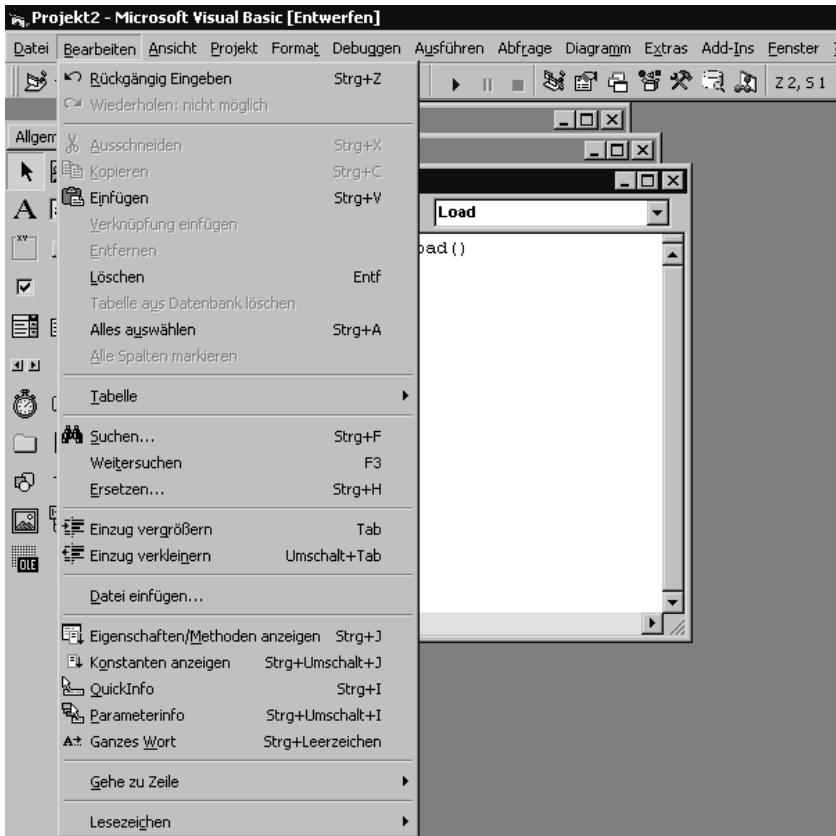


Abbildung 2.6:
Menü Bearbeiten

2.2.4 Menü Ansicht

Über die Menü-Befehle des Menüs ANSICHT (Abbildung 2.7) können Sie die Darstellung und Anzeige der Fenster in der Entwicklungsumgebung einstellen. Es stehen Menü-Befehle zum Öffnen des Codefensters oder der Objektansicht sowie zum Ein- und Ausblenden der verschiedenen Hilfsfenster zur Verfügung.

Menü Ansicht

2.2.5 Menü Projekt

Über das Menü PROJEKT (Abbildung 2.8) können Sie die dem Projekt zugehörigen Dateien und Objekte verwalten. Hier können Sie neue Komponenten wie *Formulare*, *Module*, *Klassen* etc. den offenen Projekten hinzufügen oder aus ihnen entfernen.

Menü Projekt

Mit dem Menü-Befehl VERWEISE... können Sie Bibliotheken und Anwendungen auswählen, deren Funktionen und Objekte Sie in Ihrem Projekt verwenden wollen. Es wird dann ein Verweis auf die Objektbibliothek dieser Anwendung angelegt. Nach Auswahl eines Eintrags aus dieser Liste stehen Ihnen alle Funktionen und Objekte der Bibliothek oder Anwendung in Ihrem aktuellen Projekt zur Verfügung.

**Menü-Befehl
VERWEISE...**

2 Workshop: Die Entwicklungsumgebung

Abbildung 2.7:
Menü Ansicht

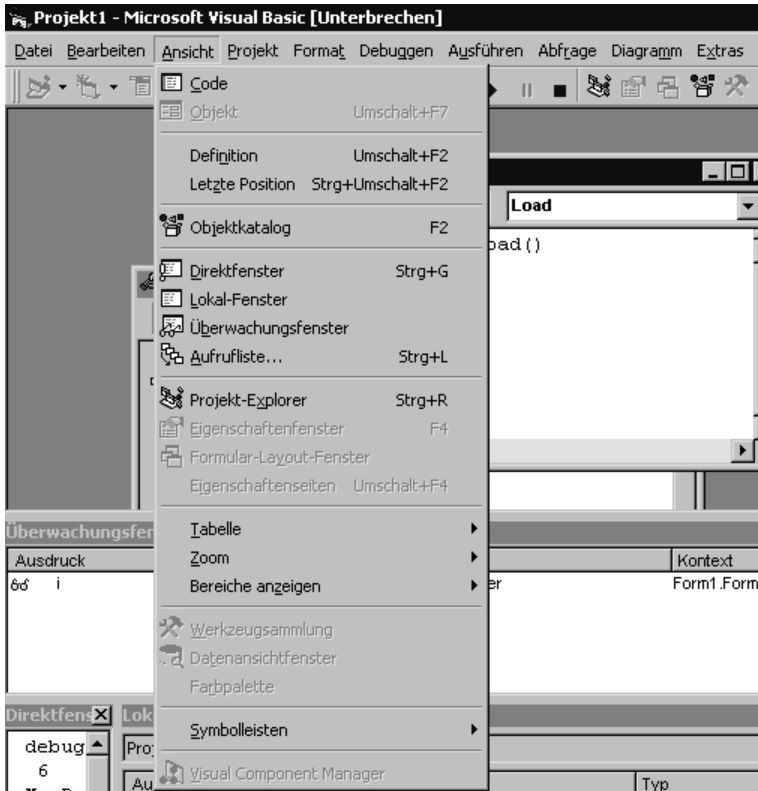
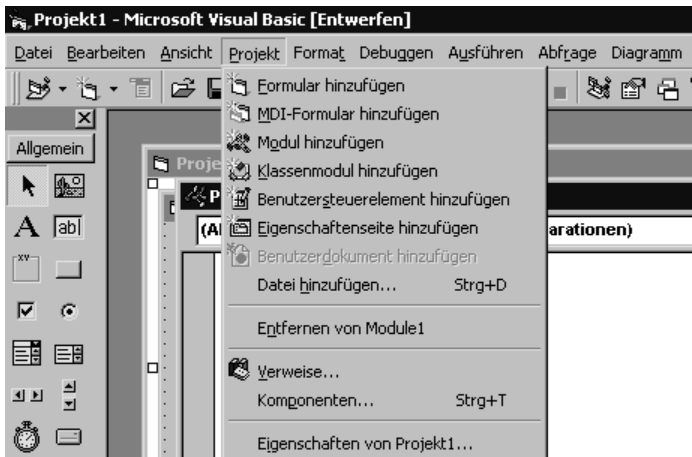


Abbildung 2.8:
Menü Projekt



**Menü-Befehl
KOMponenten...**

Der Menü-Befehl **KOMponenten...** ermöglicht es Ihnen schließlich, weitere Steuerelemente und Objekte in Ihr Projekt aufzunehmen.

Der Menü-Befehl EIGENSCHAFTEN VON PROJEKT1... öffnet einen Dialog, in welchem alle relevanten, projektbezogenen Eigenschaften eingestellt werden können.

**Menü-Befehl
EIGENSCHAFTEN VON
PROJEKT1...**

2.2.6 Menü Format

Mit dem Menü FORMAT (Abbildung 2.9) erreichen Sie eine ganze Reihe von Befehlen, die es Ihnen erlauben, die Anordnung der Steuerelemente zu verändern und sie auszurichten.

Menü Format

Dies bezieht sich vor allem auf Gruppen von Steuerelementen. Ein einzelnes Steuerelement lässt sich mit den Menü-Befehlen lediglich am Raster ausrichten.

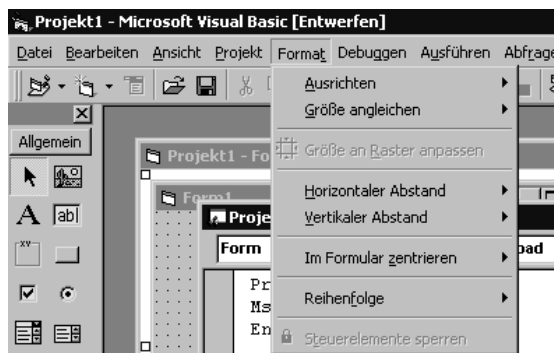


Abbildung 2.9:
Menü Format

2.2.7 Menü Debuggen

Das Menü DEBUGGEN (Abbildung 2.10) dient dem Überwachen und Testen einer Anwendung. Es beinhaltet verschiedene Menüpunkte zur Prozessablaufsteuerung, die in der Regel erst nach einer Unterbrechung des Programmablaufs wirksam werden.

Menü Debuggen



Abbildung 2.10:
Menü Debuggen



2.2.8 Übung: Programm Debuggen

Laden Sie das der Übung zugehörige Projekt in die Entwicklungsumgebung. Starten Sie eine Debug-Sitzung. Setzen Sie einen Haltepunkt. Arbeiten Sie im Einzelschritt-Modus und benutzen Sie die Funktionen um Variablen zu betrachten.

Lösung

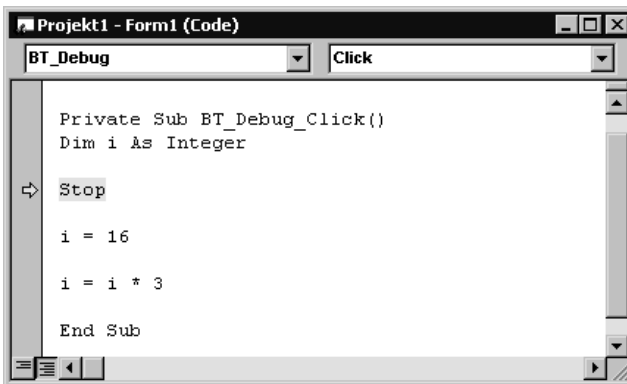
Projekt starten Nachdem Sie das Projekt geladen haben, können Sie es direkt starten. Benutzen Sie hierfür die Funktionstaste **[F5]**. Sie sehen die Oberfläche aus Abbildung 2.11.

Abbildung 2.11:
Das Programm
wurde gestartet



Nachdem Sie die Schaltfläche *Starten* ausgewählt haben, öffnet sich ein Code-Fenster (Abbildung 2.12).

Abbildung 2.12:
Programmlauf
wurde durch
Stop-Anweisung
unterbrochen



Programmlauf unterbrechen

Wenn Sie ein Programm debuggen möchten, so müssen Sie den Programmlauf unterbrechen, denn nur dann haben Sie die Möglichkeit auf den Programm-
lauf Einfluss zu nehmen bzw. einen aktuellen Status der Programmzustände zu erfahren.

Haltepunkt setzen

In Abbildung 2.12 wurde der Programm-
lauf durch eine *Stop*-Anweisung unterbrochen. Eine weitere Möglichkeit ist das Setzen eines Haltepunktes. Hierzu müssen Sie den Eingabecursor in die Zeile stellen, in welcher Sie anhalten möchten. Dann drücken Sie die Taste **[F9]** oder wählen den Menüpunkt **DEBUGGEN-> HALTEPUNKT EIN/AUS**.

Sie erkennen den Haltepunkt im Code-Fenster am roten Punkt links neben der Programmzeile (Abbildung 2.13).

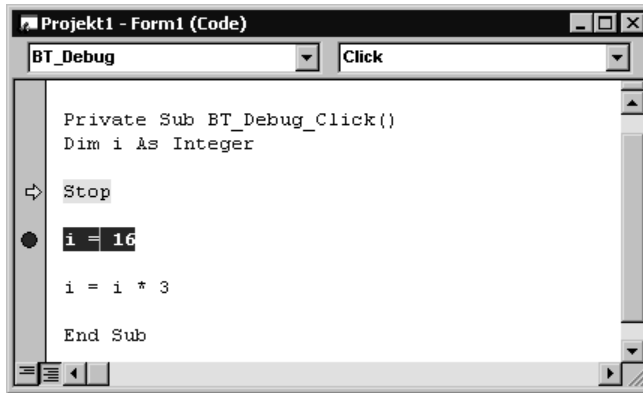


Abbildung 2.13: Haltepunkt wurde gesetzt

Durch Betätigen der Taste **F5** wird der Programmlauf fortgesetzt. Der nächste Stop ist auf dem gesetzten Haltepunkt.

Sie können im Haltemodus jede Anweisung im Code einzeln ausführen und das Ergebnis dann sofort analysieren. Sehr hilfreich sind hier das *Direktfenster*, das *Lokalfenster* und das *Überwachungsfenster* (Abbildung 2.14).

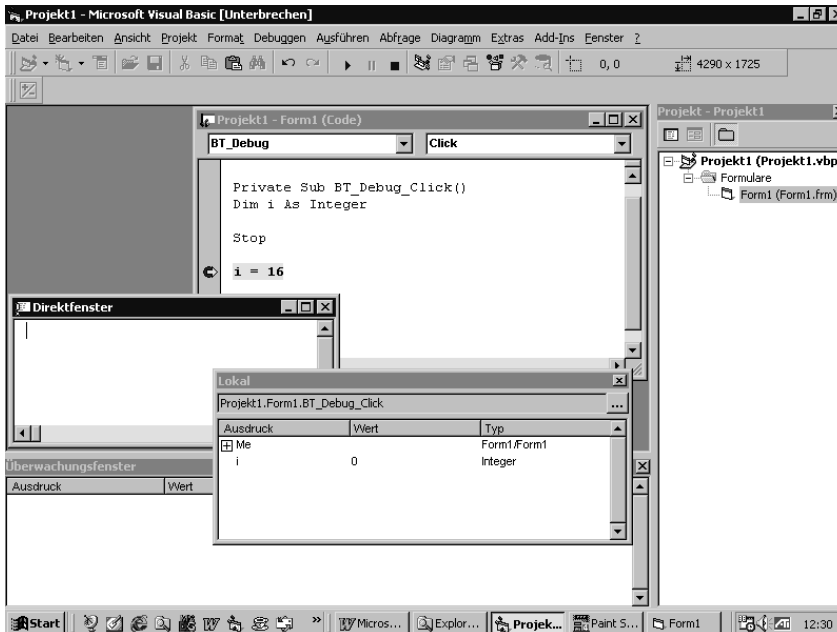
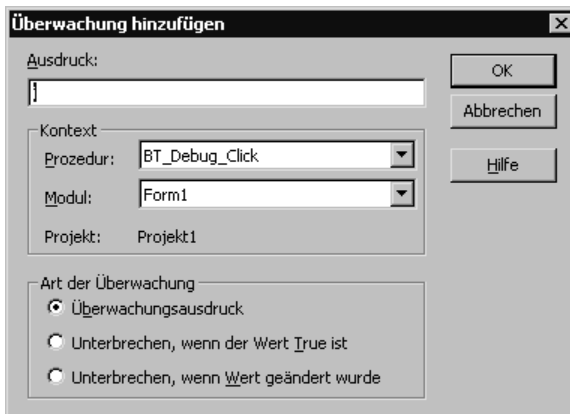


Abbildung 2.14: Direktfenster, Lokalfenster und Überwachungsfenster

2 Workshop: Die Entwicklungsumgebung

- das Lokalfenster** Im Lokalfenster werden alle Variablen der aktuellen Prozedur mit ihren Werten angezeigt. Es wird nur dann aktualisiert, wenn der Programmablauf unterbrochen wurde.
- das Überwachungsfenster** Gleiches gilt für das Überwachungsfenster. Im Gegensatz zum Lokalfenster werden im Überwachungsfenster allerdings nur diejenigen Variablen angezeigt, die explizit von Ihnen zur Überwachung ausgewählt wurden. Dies kann durch das Menü **DEBUGGEN->ÜBERWACHUNG HINZUFÜGEN** erfolgen.
- das Direktfenster** Das Direktfenster schließlich erlaubt Ihnen die Eingabe von Programmanweisungen und deren Ausführung durch Betätigen der Taste **[ENTER]**. Außerdem können mit der Anweisung *Debug.Print* Ausgaben im Direktfenster angezeigt werden.
- Überwachung hinzufügen** Um eine Variable zu überwachen, muss Visual Basic über das Menü **DEBUGGEN->ÜBERWACHUNG HINZUFÜGEN...** dazu veranlasst werden. Setzen Sie die Eingabemarke auf eine Variable. Hat die Variable nur einen Buchstaben, wie im Beispiel, so können Sie die Eingabemarke direkt vor oder nach den Variablennamen stellen. Rufen Sie über den Menü-Befehl den Dialog *Überwachung Hinzufügen* (Abbildung 2.15) auf und bestätigen Sie ihn mit *Ok*.

Abbildung 2.15:
Dialog
Überwachung
hinzufügen

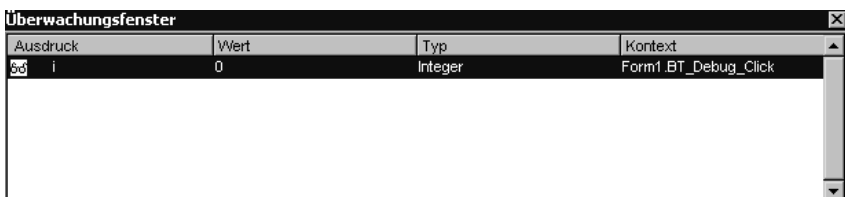


Das Überwachungsfenster zeigt jetzt die Variable an. Sie hat den Wert 0 (Abbildung 2.16). Die Programmzeile weist der Variablen aber den Wert 16 zu.



Das Programm wird durch einen Haltepunkt gestoppt, bevor die Programmzeile, auf der sich der Haltepunkt befindet, ausgeführt wird.

Abbildung 2.16:
Wert der Variablen
auf dem
Haltepunkt



Um das Programm im Einzelschrittmodus weiterzuführen, benutzen Sie die Funktionstaste **F8** oder den Menü-Befehl **DEBUGGEN->EINZELSCHRITT**. In Abbildung 2.17 können Sie sehen, dass sich der Wert der Variablen *i* im Überwachungsfenster geändert hat.

Einzelschrittmodus

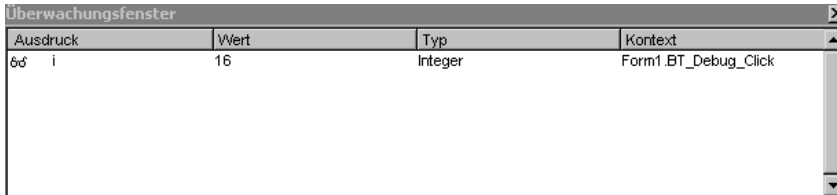


Abbildung 2.17: Wert der Variablen nach Ausführen der Programmzeile im Einzelschrittmodus

Sie können jetzt das Programm durch mehrfaches Betätigen der Funktionstaste **F8** oder durch einmaliges Betätigen der Funktionstaste **F5** zu Ende laufen lassen.

Sie haben jetzt die Grundtechniken des Debuggens mit Visual Basic kennen gelernt. Der Debugger von Visual Basic ist aber ein weitaus mächtigeres Werkzeug, als bisher gezeigt wurde. Es lohnt sich auf jeden Fall, hierzu die Online-Hilfe von Visual Basic zusätzlich zu konsultieren.



2.2.9 Menü Ausführen

Im Menü **AUSFÜHREN** (Abbildung 2.18) sind alle Befehle zusammengefasst, die benötigt werden, um ein Programm in der Entwicklungsumgebung zu starten.

Menü Ausführen

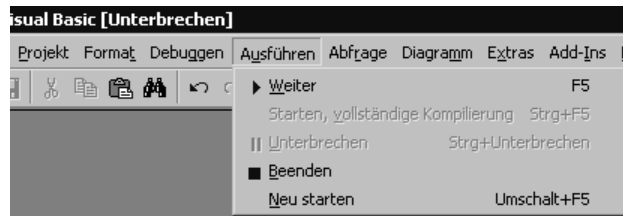


Abbildung 2.18: das Menü Ausführen

2.2.10 Menü Extras

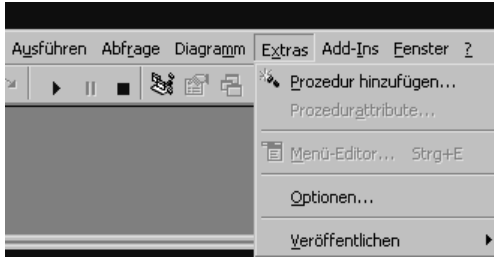
Das Menü **EXTRAS** (Abbildung 2.19) beinhaltet unter anderem den *Menü-Editor*. Ein geöffnetes und aktives Formularfenster kann mit Hilfe dieses Editors mit einem Menü versehen werden.

Menü Extras

Ein weitaus wichtigerer Punkt im Menü **EXTRAS** ist jedoch das Menü **OPTIONEN**. Dieser Menü-Befehl öffnet den Dialog *Optionen*, über welchen globale Einstellungen der Entwicklungsumgebung vorgenommen werden können.

Visual Basic Optionen

Abbildung 2.19:
Menü Extras



2.2.11 Menü Add-Ins

Menü Add-Ins Neben den von anderen Windows-Programmen her bekannten Menüpunkten *Fenster* und *?* gibt es noch den Menüpunkt *Add-Ins*.

Hierin sind alle installierten und freigeschalteten Add-Ins aufgelistet. Der *Add-In-Manager* in diesem Menü regelt diese Auflistung.

Add-Ins sind eigenständige Programme, die in die Entwicklungsumgebung auch nachträglich integriert werden können. Die Anzahl der mitgelieferten Add-Ins hängt von Ihrer Edition ab. Weitere Add-Ins können von anderen Herstellern zugekauft und in die Visual Basic-Entwicklungsumgebung integriert werden.

2.3 Symbolleisten

Symbolleisten Die Symbolleisten stellen ein *Mittelding* zwischen Menü-Befehlen und Tastaturkürzeln dar. Sie ermöglichen Ihnen, einen Menü-Befehl mit einem Mausklick auf ein Symbol auszuführen. Diese Symbole sind, nach Themenbereichen geordnet, in so genannten Leisten untergebracht. Sie befinden sich unterhalb der Menüleiste.

Sie haben auch die Möglichkeit, den Inhalt der Symbolleisten anzupassen oder gar eigene Symbolleisten zu entwerfen. Hierzu dient der Menü-Befehl ANSICHT>SYMBOLLEISTEN->ANPASSEN... .

2.4 Das Eigenschaftfenster

das Eigenschaftfenster

Im *Eigenschaftfenster* (Abbildung 2.20) werden während der Programmentwicklung die Eigenschaften von Objekten (Formulare, Klassen, Steuerelemente) angezeigt. Sie können dort auch verändert werden. Das *Eigenschaftfenster* zeigt immer die Eigenschaften des aktuell markierten Objekts an.

Falls das Eigenschaftfenster nicht sichtbar ist, kann es über das Menü ANSICHT->EIGENSCHAFTENFENSTER, über die Taste **F4** oder über die Symbolleiste aktiviert werden.

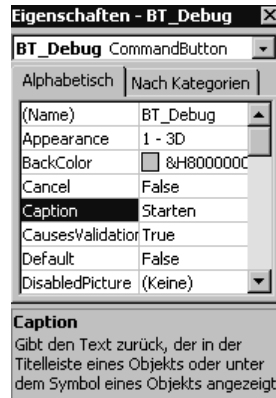


Abbildung 2.20:
Die Eigenschaften der Schaltfläche BT_Debug im Eigenschaftfenster

Das Eigenschaftfenster besteht aus den Teilen *Objekt-Listbox* und der *Eigenschaften-Liste*.

Objekt-Listbox

In der *Objekt-Listbox*, die sich direkt unterhalb des Fenstertitels befindet, sind alle Objekte des aktuell aktiven Moduls eingetragen. Falls nur ein Objekt markiert ist, wird der Name des markierten Objekts in der *Textbox* angezeigt. Mit der *DropDown-Liste* kann, ohne das Eigenschaftfenster zu verlassen, auf jedes andere Objekt des aktiven Moduls gewechselt werden.

alle Objekte des aktiven Moduls

Eigenschaftenliste

Die *Eigenschaftenliste* ist eine Tabelle mit zwei Spalten. Die linke Spalte enthält die Eigenschaften des markierten Objekts. Die rechte Spalte enthält die aktuellen Werte dieser Eigenschaften.

Eigenschaften und Werte



Abbildung 2.21:
Eigenschaften nach Kategorien geordnet

Über die Registerkarten der Liste können die Eigenschaften alphabetisch oder, wie in Abbildung 2.21 sichtbar, nach Kategorien geordnet angezeigt werden.

nach Kategorien geordnet



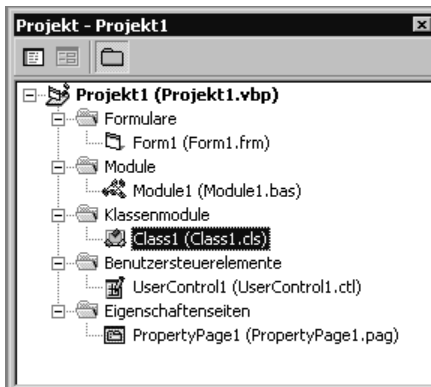
Das Setzen von Eigenschaften ist in Visual Basic der halbe Weg zum fertigen Programm. Unterschätzen Sie nicht die Macht der Eigenschaften. Viele Zeilen Code meiner Programme zeigten sich im Nachhinein als überflüssig, nachdem ich die Eigenschaften der Objekte besser definierte.

2.5 Der Projekt-Explorer

**alle Dateien
des/
Projekts/Projekte**

Der *Projekt-Explorer* liefert eine Liste aller im Projekt oder der Projektgruppe verwendeten Module. Abbildung 2.22 zeigt dabei nur eine kleine Auswahl der möglichen Projektdateien. Ob normale *Formulare* oder *MDI-Formen*, *Module*, *Klassen*, *Steuerelementdateien* oder *Eigenschaftenseiten*, im Projektfenster sind alle Projektdateien aufgelistet.

Abbildung 2.22:
Alle Dateien eines
Projekts werden im
Projekt-Explorer
verwaltet



Vom *Projektfenster* aus kann das *Formularfenster* oder das *Code-Fenster* eines Moduls geöffnet werden. Es muss lediglich das entsprechende Modul in der Liste markiert werden. Jetzt wird entweder das Symbol *Code anzeigen* angewendet, um das zugehörige *Code-Fenster* zu öffnen, oder es wird das Symbol *Objekt anzeigen* angewendet, um das zugehörige *Formularfenster* zu öffnen.

einfach anklicken

Wird ein Modul im Projektfenster doppelt angeklickt, so wird für Formulare automatisch das *Formularfenster*; für Module automatisch das *Code-Fenster* geöffnet.

2.5.1 Kontextmenüs des Projekt-Explorers

Kontextmenüs

Wird über dem Projektfenster die rechte Maustaste gedrückt, so öffnet sich ein Kontextmenü (Abbildung 2.23).

In diesem Menü (Abbildung 2.23) können dem Projekt weitere Projekt-Module hinzugefügt werden.

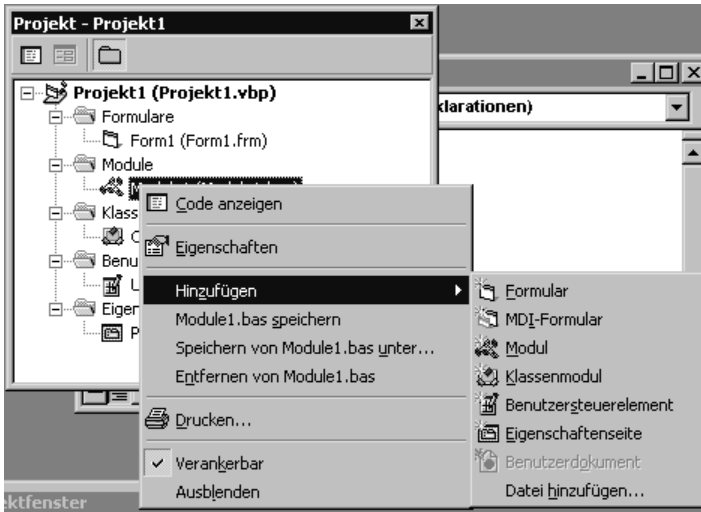


Abbildung 2.23:
Das Kontextmenü
des Projekt-
Explorers

Falls Sie beim Betätigen der rechten Maustaste nicht die Menüs aus Abbildung 2.23 gesehen haben, so befanden Sie sich wahrscheinlich mit dem Mauszeiger über dem Eintrag *Projekt* des Projekt-Explorers. In diesem Fall zeigt der Projekt-Explorer nämlich das Projekt-Kontextmenü aus Abbildung 2.24.

Projekt-Kontextmenü



Abbildung 2.24:
Das Projekt-
Kontextmenü des
Projekt-Explorers

Wenn Sie die beiden Menüs miteinander vergleichen, werden Sie feststellen, dass die Menü-Befehle, die sich um das Speichern und Entfernen von Projektmodulen drehen, jetzt auf das ganze Projekt beziehen, statt wie vorher nur auf ein einzelnes Modul.

Projekt speichern

Zudem können Sie den *Paket- und Weitergabeassistent* aufrufen, sofern Sie Ihre Software ausliefern möchten. Des Weiteren ist es möglich, für eine Projektgruppe das *Start-Projekt* festzulegen. Das Startprojekt ist jenes, welches bei Eingabe der Taste F5 gestartet wird. Diese Auswahl ist allerdings nur dann sinnvoll, wenn tatsächlich mehr als ein Projekt geladen ist.

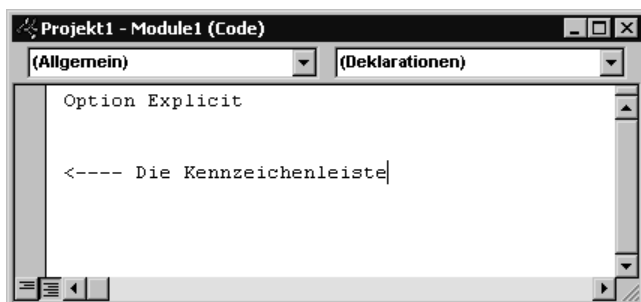
Paket- und Weitergabeassistent

2.6 Das Code-Fenster

Code-Fenster = Editor

Das *Code-Fenster*, auch Editor genannt, präsentiert sich mit einigen nützlichen Eigenschaften. Zuerst wird Ihnen wohl der senkrechte Strich am linken Rand auffallen. Dies ist die so genannte Kennzeichenleiste (Abbildung 2.25).

Abbildung 2.25:
Das Code-Fenster
mit Kennzeichen-
leiste



Kennzeichenleiste

In der Kennzeichenleiste können Lesezeichen und Haltepunkte angezeigt und gesetzt werden. Im Abschnitt über das Debuggen konnten wir dies bereits sehen.

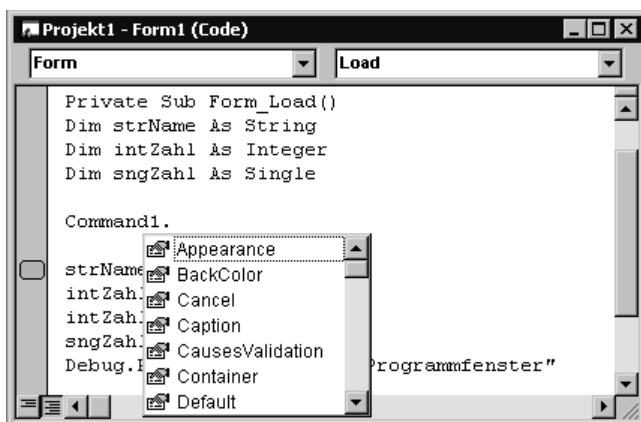
Das Code-Fenster hat aber einige weitere Funktionen, die das Programmieren sehr erleichtern:

Automatische Anweisungsergänzung

Anweisungsergänzung

Wenn Sie im Code auf ein Objekt mit einer Methode oder Eigenschaft Bezug nehmen, so wird Ihnen nach Eingabe des Punktes eine *Drop-Down*-Liste angezeigt, die alle Methoden und Eigenschaften des Objekts enthält (Abbildung 2.26).

Abbildung 2.26:
Automatische
Anweisungsergänzung
im Code-Fenster



Mit den Pfeiltasten oder mit der Maus wählen Sie Ihr gewünschtes Element. Wenn Sie dann die Leertaste oder die Tabulatortaste drücken, wird das gewählte Element automatisch rechts hinter dem Punkt eingetragen.

Dies erleichtert Ihnen die Suche nach den verfügbaren Elementen des Objekts und verhindert darüber hinaus Schreibfehler. Die *Automatische Anweisungsergänzung* kann im Dialog *Optionen* unter *Editor->Elemente automatisch auflisten* aktiviert werden.

keine Schreibfehler

QuickInfo

Dies ist ein weiteres Hilfsmittel, um Fehleingaben zu verhindern. Wenn im Dialog *Optionen* in *Editor ->Automatische QuickInfo* aktiviert ist, werden Sie beim Eingeben eines Funktionsaufrufs automatisch von den zugehörigen Parametern in Kenntnis gesetzt (Abbildung 2.27).

QuickInfo

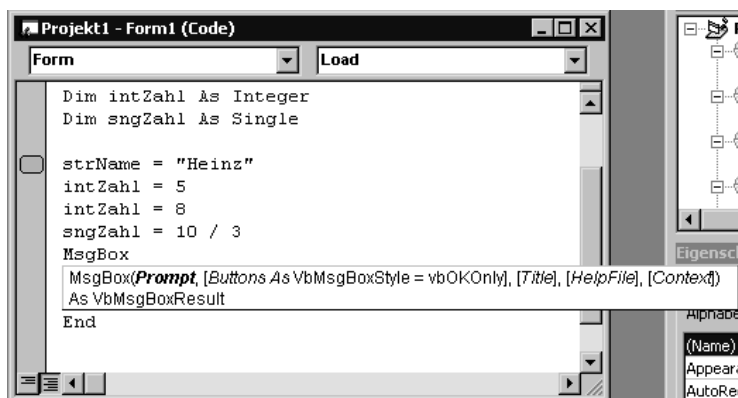


Abbildung 2.27: Automatische QuickInfo zeigt die Parameter der Methode MsgBox

Es wird ein Infowindow unter Ihrer aktuellen Zeile aufgeblendet, welches die Funktionssyntax anzeigt. Das erste Argument wird fett dargestellt. Wenn zu diesem ein Wert eingegeben wurde, wechselt die Markierung auf das zweite Argument, und so weiter. Es kann Ihnen somit nicht mehr passieren, dass Sie zu wenig Argumente oder einen falschen Typ an eine Funktion übergeben.

2.7 Die Werkzeugsammlung und das Formularfenster

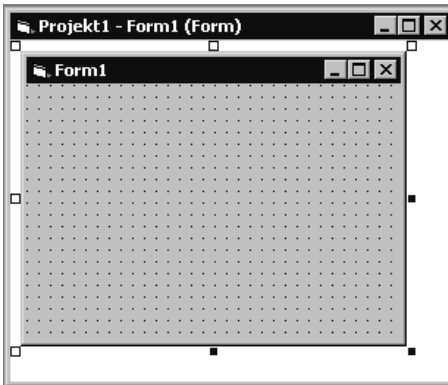
Zwei weitere, nicht wegzudenkende Elemente der Visual Basic-Entwicklungsfläche sind die *Werkzeugsammlung*, auch Toolbox genannt, und das *Formularfenster*.

Mit diesen zwei Werkzeugen wird im Wesentlichen die Oberfläche eines Programms erstellt. Ich habe dabei bewusst das Wort »programmiert« nicht verwendet, denn die Erstellung einer Oberfläche ist in Visual Basic nichts anderes als das Zusammenstellen von mehreren Objekten im Formularfenster.

Oberflächen-design

Ein Formularfenster kann für jedes Programm-Objekt mit grafischer Oberfläche (beispielsweise eine Form) geöffnet werden (Abbildung 2.28).

Abbildung 2.28:
Das Objekt Form1
im Formularfenster
(Designer)

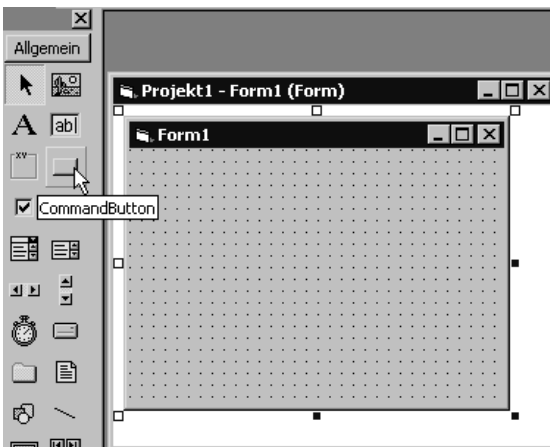


2.7.1 Die Werkzeugsammlung – Objekte hinzufügen

Die Werkzeugsammlung ist das einzige Instrument, mit dem man neue Objekte in ein Formularfenster aufnimmt. Dabei können Sie Objekte anhand ihrer grafischen Darstellung in der Werkzeugsammlung oder anhand der in Abbildung 2.29 dargestellten QuickInfo identifizieren.

Um die QuickInfo zu erhalten, müssen Sie lediglich mit dem Mauszeiger eine kurze Zeitspanne über der entsprechenden Grafik verharren.

Abbildung 2.29:
Ein Objekt in der
Werkzeugsamm-
lung identifizieren



Um eine Schaltfläche auf der Form zu platzieren, wird diese, wie in Abbildung 2.29 angedeutet, in der Werkzeugsammlung doppelt angeklickt. Daraufhin wird das neue Objekt, in diesem Fall eine Schaltfläche, auf dem Formular (Objekt *Form1*) mittig platziert.

Um ein Objekt im Formularfenster an eine andere Position zu bewegen, wird es mit der Maus bei gedrückter Taste gezogen. Die Größe der Objekte kann ebenfalls per Drag&Drop geändert werden. Alle weiteren Eigenschaften werden über das Eigenschaftenfenster verändert.

**Position und
Größe von
Objekten ändern**

3

Workshop: Variablen und Konstanten

In diesem Kapitel wird das Thema Variablen und Konstanten in Visual Basic vorgestellt. Es werden einige Übungen zu folgenden Themen gestellt und gelöst:

- ▶ Variablendeklaration
- ▶ Datentypen von Variablen
- ▶ Gültigkeitsbereiche von Variablen
- ▶ Verwendung von Konstanten

Vor den Übungen zum jeweiligen Thema wird immer erst die Theorie der Thematik in aller Kürze vorgestellt.

3.1 Variablen

Variablen sind Platzhalter. Sie werden während der Ausführung eines Programms verwendet, um Werte temporär zu speichern. Der gespeicherte Wert einer Variablen kann während der Programmlaufzeit unbegrenzt oft verändert werden.

Variablen sind Platzhalter

Wenn Sie in einem Programm einen Namen und ein Geburtsdatum speichern, so werden Sie dafür normalerweise unterschiedliche Variablen verwenden. Die beiden Variablen unterscheiden sich durch ihren Datentyp, d.h. durch die Art der Werte, die in ihr gespeichert werden können.

Datentypen

Ein Name wird beispielsweise in einer Variablen gespeichert, die Zeichenketten verarbeiten kann. Ein Datum hingegen sollte möglichst in einer Variablen gespeichert werden, die mit diesem Wert auch etwas anfangen kann.

3.1.1 Die Deklaration von Variablen

Wenn Sie eine Variable verwenden möchten, so müssen Sie dies Visual Basic mitteilen, d.h. Sie müssen in Ihrem Programm bekannt machen, dass durch eine bestimmte Zeichenfolge eine Variable bezeichnet wird. Dieser Vorgang des Bekanntmachens wird als Deklaration bezeichnet. Bei der Deklaration einer Variablen wird auch deren Datentyp festgelegt.

Variable bekannt machen

Visual Basic kennt für die Deklaration einer Variablen zwei Wege:

- ▶ Die *implizite Deklaration*
- ▶ Die *explizite Deklaration*

3.1.2 Die implizite Deklaration

automatische Deklaration

Eine *implizite Deklaration* wird von Visual Basic automatisch dann durchgeführt, wenn in Ihrem Programm ein neuer Name auftaucht, der nicht bereits durch eine andere Bedeutung belegt ist. Visual Basic interpretiert in diesem Fall den neuen Namen als neue Variable und verwendet automatisch einen *Variant* als Datentyp.



Vorsicht bei Schreibfehlern

Es scheint auf den ersten Blick sehr praktisch zu sein, Variablen automatisch von Visual Basic deklarieren zu lassen. Bei genauerer Untersuchung wird jedoch klar, dass diese Art der Deklaration sehr fehleranfällig ist:

Durch einen einfachen Schreibfehler wird automatisch eine neue Variable generiert. Dies führt im Programm mit Sicherheit zu einem ungewollten Verhalten oder einem falschen Ergebnis. Verwenden Sie daher die *implizite Deklaration* von Variablen grundsätzlich nicht.

Das folgende Beispiel zeigt einen Fehler dieser Art:

```
Preis = 100
Prozent = 16
Mehrwertsteuer = Preis / 100 * Ptozent
MsgBox ("Mehrwertsteuer: " & Mehrwertsteuer)
```

Lassen Sie dieses Programm laufen, so erhalten Sie als Ergebnis den Betrag 0. Dies ist aber leider falsch. Der Grund hierfür ist ein Schreibfehler bei der Berechnung.

Visual Basic kann in einer solchen Situation nicht feststellen, dass der Name einer existierenden Variablen, nämlich *Prozent*, falsch geschrieben wurde. Daher wird die Variable *Prozent* automatisch von Visual Basic angelegt und mit 0 initialisiert. Passiert Ihnen dies in einem größeren Projekt, können Sie mit einer aufwändigen Fehlersuche beginnen.

3.1.3 Die explizite Deklaration

Option Explicit

Um das Risiko falsch geschriebener Variablenamen auszuschließen, müssen Sie lediglich im Deklarationsteil einer *Klasse*, *Form* oder eines *Standardmoduls* die Anweisung `Option Explicit` einfügen. Sobald dies geschehen ist, lässt Visual Basic die *implizite Deklaration* von Variablen nicht mehr zu. Stattdessen wird ein neuer Name bei der Übersetzung (Kompilieren) des Programms mit der Fehlermeldung in Abbildung 3.1 quittiert.

explizite Deklaration als Grundeinstellung

Die *explizite Deklaration* kann auch als Grundeinstellung für neue Projekte definiert werden. Öffnen Sie dazu den Dialog `OPTIONEN...` (Abbildung 3.2), der über das Menü `EXTRAS->OPTIONEN` erreicht wird.

Auf der Registerkarte `EDITOR` wird das Kontrollkästchen *Variablendeklaration erforderlich* markiert. Ab jetzt wird von Visual Basic, beim Anlegen neuer Module, die Deklaration `Option Explicit` automatisch eingefügt.

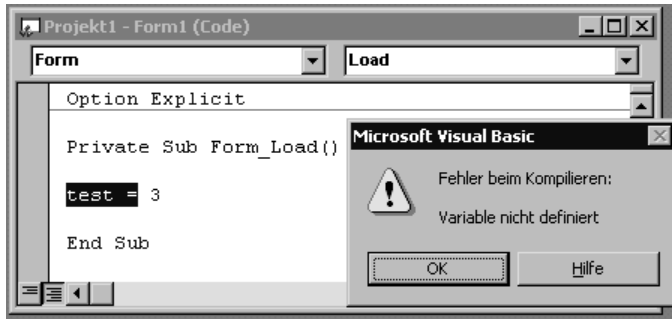


Abbildung 3.1:
Variable nicht
definiert

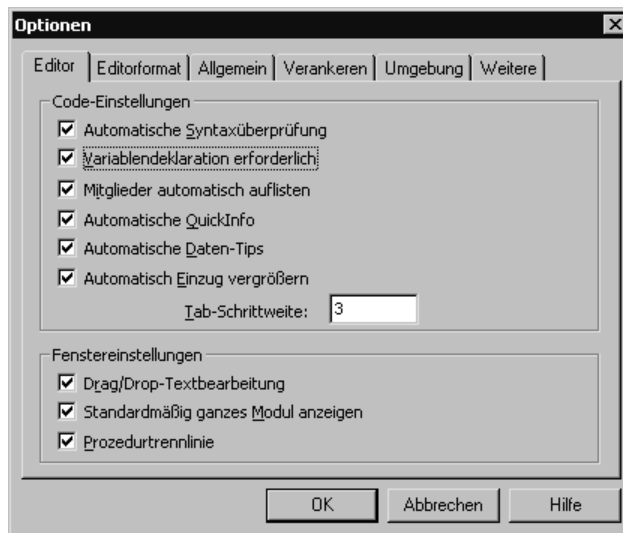


Abbildung 3.2:
Voreinstellung
Variablendeklara-
tion erforderlich

Bei der *expliziten Deklaration* einer Variablen muss ein Name sowie ein Datentyp angegeben werden.

Der Variablenname wird auch *Bezeichner* genannt und muss folgende Bedingungen erfüllen:

- ▶ Das erste Zeichen ist ein Buchstabe.
- ▶ Er darf nicht länger als 255 Zeichen sein.
- ▶ Punkte, Leerzeichen und Typenkennzeichen (\$, @, %, &, !, #) sind nicht erlaubt.
- ▶ Der Variablen- bzw. Bezeichnername darf keinem Namen entsprechen, der als Schlüsselwort von Visual Basic verwendet wird.

Die Anweisungen *Dim*, *ReDim*, *Public*, *Private* oder *Static* werden dem Variablennamen vorangestellt.

Regeln für Variablennamen

3.1.4 Syntax einer Variablendeklaration

Syntax Deklarationsschlüsselwort *Varname*[([*Feldgröße*])] [As [*New*] *Datentyp*]

► Deklarationsschlüsselwort

Dim: Deklariert eine Variable auf Modul- oder Prozedurebene. Bei Deklaration im allgemeinen Teil eines Moduls ist die Variable für alle Prozeduren dieses Moduls verfügbar. Wird die Variable hingegen innerhalb einer Prozedur deklariert, so ist sie auch nur in dieser verfügbar.

Static: Deklariert eine Variable innerhalb einer Prozedur, die ihren Wert bis zum nächsten Aufruf derselben Prozedur beibehält.

Public: Durch die Deklaration einer Variablen mit dem Schlüsselwort *Public* im allgemeinen Teil eines Moduls wird eine öffentliche Variable definiert. Diese ist im kompletten Programm, d.h. in allen Formularen und Modulen des Projektes verfügbar.

Private: Eine auf Modulebene mit *Private* deklarierte Variable ist in dem Modul verfügbar, in welchem sie deklariert wurde.

ReDim: Wird auf Prozedurebene verwendet, um die Indexgrenzen einer dynamischen Feldvariablen neu zu setzen. Der bisherige Inhalt der Feldvariablen wird dabei gelöscht. Durch das zusätzliche Schlüsselwort *Preserve* ist es möglich, den bisherigen Feldinhalt beizubehalten.

► Varname

Varname steht für den Namen der zu deklarierenden Variablen. Er muss den Namenskonventionen für Variablen entsprechen.

► Feldgröße

Durch Angabe einer *Feldgröße* wird die Variable als Feld (Array) deklariert. Die *Feldgröße* kann eine einzelne Zahl sein oder ein Bereich mit Unter- und Obergrenze.

► New

Die Verwendung des Schlüsselwortes *New* weist Visual Basic an, eine neue Instanz eines Objektes anzulegen.

► Datentyp

Über *Datentyp* wird festgelegt, welchen der in Frage kommenden Datentypen die Variable erhalten soll.

3.2 Datentypen und Wertebereiche

Visual Basic unterscheidet zunächst zwei grundlegende Datentypen:

- ▶ *Nummerische Datentypen*: Werden verwendet, um Zahlen zu speichern, die errechnet werden und/oder für Berechnungen innerhalb des Programms benötigt werden.
- ▶ *Alphanummerische Datentypen*: Werden verwendet, um Zeichenketten, also Wörter oder Buchstaben zu speichern.

Zudem wird in einigen Programmiersprachen, so auch in Visual Basic, ein Datentyp angeboten, der nur die Werte *Richtig*, oder *Falsch* annehmen kann. Dieser Datentyp heißt *Boolean* und wird meist dann verwendet, wenn eine Ja/Nein-Entscheidung zu fällen ist.

Visual Basic kennt zusätzlich noch einen Datentyp: den *Variant*. Dieser Datentyp passt sich dem gespeicherten Inhalt an, d.h. wird eine Zahl zugewiesen, so verhält sich der *Variant* wie eine numerische Variable, wird hingegen eine Zeichenkette zugewiesen, so verhält er sich wie eine alphanummerische Variable.

Nun werden Sie sich vielleicht fragen, warum es überhaupt andere Datentypen gibt, wenn der *Variant* doch jeden Datentyp abbilden kann.

Wird der Datentyp *Variant* verwendet, so wird die in Visual Basic integrierte Typprüfung weitgehend ausgehebelt. Sollten Sie in Ihrem Programm einer numerischen Variablen (des Datentyps *Variant*) eine Zeichenkette zuweisen, so kann Visual Basic diesen Fehler beim Übersetzen des Programms nicht feststellen.

Tabelle 3.1 zeigt die wichtigsten Datentypen von Visual Basic, den zugehörigen Wertebereich und das entsprechende Schlüsselwort, welches bei der Variablen-deklaration benutzt werden muss.



Datentypen und Wertebereiche

Datentyp	Wertebereich	Schlüsselwort
Boolean	True oder -1 entspricht wahr False oder 0 entspricht falsch	Boolean
Byte	0 – 255	Byte
String variabler Länge	0 – ca. 2 Milliarden Zeichen	String
String fester Länge	0 – ca. 65.500 Zeichen	String * x
Integer	-32.768 bis +32.767	Integer
Longinteger	-2.147.483.648 bis +2.147.483.647	Long
Fließkomma einfache Genauigkeit	-3,402823E38 bis -1,401298E-45 und 1,401298E-45 bis 3,402823E38	Single

Tabelle 3.1:
Visual Basic-Daten-
typen und deren
Wertebereich

Datentyp	Wertebereich	Schlüsselwort
Fließkomma doppelte Genauigkeit	-1,79769313486232E308 bis -4,94065645841247E-324 und 4,94065645841247E-324 bis 1,79769313486232E308	Double
Currency	-922.337.203.685.477,5808 bis 922.337.203.685.477,5807	Currency
Date	1. Januar 100 bis 31. Dezember 9999	Date
Variant mit numerischem Wert	Jeder numerische Wert im Wertebereich einer Double-Variablen	Variant
Variant mit Stringwert	0 bis ca. 2 Milliarden	Variant

Anhand der Tabelle können Sie jetzt problemlos die Deklaration verschiedener Variablentypen durchführen. Sie müssen dazu lediglich wissen, für welchen Zweck die Variable benötigt wird. Dann suchen Sie in der Tabelle einen Datentyp, der für diesen Zweck passt, und schreiben die Definition im Visual Basic-Programmmeditor, je nach Gültigkeitsbereich, an die entsprechende Stelle.



3.2.1 Übung: Deklaration von Variablen

Deklariert Sie Variablen für folgenden Verwendungszweck:

1. Speichern eines Geldbetrags, der 20000 nicht übersteigt.
2. Speichern eines Namens.
3. Speichern eines Geburtsdatums.
4. Speichern des Geschlechts einer Person.
5. Speichern des Ergebnisses einer Division.
6. Speichern Sie einen Wert bis zum nächsten Aufruf einer Funktion.

Bitte verwenden Sie für die Lösung der Übung *nicht* den Datentyp Variant. Alle Informationen, die Sie zur Lösung dieser Übung benötigen, finden Sie in der Syntaxbeschreibung der Variablendeklaration und in Tabelle 3.1.

Lösung

1. Speichern eines Geldbetrags, der 20000 nicht übersteigt.

Ein Geldbetrag ist eine numerische Variable. Daher kommen die Datentypen *Byte*, *Integer* und *Long* in Frage.

Für Beträge, die 20000 nicht übersteigen, ist der Wertebereich einer *Integer*-Variablen ausreichend. Die Deklaration sieht also folgendermaßen aus:

```
Dim intGeldBetrag As Integer*
```

Eine Variable des Typs *Long* würde aufgrund des größeren Wertebereichs mehr Speicherplatz benötigen. Allerdings fällt bei einem Rechner mit dem Speicherausbau heutiger Generation der Unterschied zwischen den beiden Variablentypen nicht ins Gewicht. Bei einer *Long*-Variablen wird allerdings seltenst eine Bereichsüberschreitung auftreten. Daher ist es durchaus zu empfehlen, für ganze numerische Variablen ausschließlich den Datentyp *Long* zu verwenden.



Die Deklaration würde in diesem Fall also wie folgt aussehen:

```
Dim lngGeldBetrag As Long
```

2. Speichern eines Namens.

Eine Name ist ein alphanummerischer Wert. Daher kommt für die Deklaration nur der Datentyp *String* in Frage. Die Lösung der Übung sieht also wie folgt aus:

```
Dim strVorname As String
```

3. Speichern eines Geburtsdatums.

Ein Datum ist weder ein einzelner numerischer, noch ein alphanummerischer Wert. Sie können zwar ein Datum für eine einfache Ausgabe als Zeichenkette speichern, wenn Sie aber mit diesem Datum rechnen möchten (z.B.: Wie viele Tage ist er älter/jünger als ich?), dann ist dies mit einer Zeichenkette schlicht nicht möglich.

Eine weitere Möglichkeit zur Speicherung eines Datum wäre die Trennung von Tag, Monat und Jahr und deren Speicherung in einzelne Variablen. Die Deklaration würde in diesem Fall folgendermaßen aussehen:

```
Dim bytTag as Byte
Dim bytMonat as Byte
Dim intJahr as Integer
```

Mit ein wenig Aufwand können in Ihrem Programm mit diesem Datum auch Berechnungen durchgeführt werden. Wenn Sie allerdings den Anfang des Abschnitts gründlich gelesen haben, werden Sie zweifellos zu folgender Lösung gelangt sein:

```
Dim dtmGeburtsdatum as Date
```

Visual Basic erspart uns durch den speziellen Datentyp *Date* alle Umwege und ermöglicht durch Funktionen der Laufzeitbibliothek auch Berechnungen mit diesem Datentyp durchzuführen.

4. Speichern des Geschlechts einer Person.

Das Geschlecht einer Person kann auf vielerlei Arten gespeichert werden. Sie können beispielsweise eine Variable des Datentyps *String* verwenden und »männlich« oder »weiblich« als Zeichenkette abspeichern.

Sie könnten ebenso eine numerische Variable verwenden, die von Ihrem Programm je nach Inhalt interpretiert wird. In diesem Fall definieren Sie selbst, welcher Wert männlich und welcher Wert weiblich bedeutet. Sie könnten beispielsweise festlegen, dass der Wert 1 für weiblich steht, der Wert 2 für männlich.

Da es sich bei dem Geschlecht allerdings um einen Wert mit nur zwei Zuständen handelt, können Sie den Datentyp *Boolean* verwenden:

```
Dim blnGeschlecht As Boolean
```



Der Variablenname *blnGeschlecht* ist neutral, d.h. Sie müssen wissen, ob der Wert *True* für weiblich oder männlich steht. Durch einen besseren Variablennamen können Sie die Bedeutung des Inhalts bereits verdeutlichen:

```
Dim blnWeiblich As Boolean
```

besserer Name

Jetzt ist es eindeutig: Steht in der Variablen *blnWeiblich* der Wert *True*, so handelt es sich um das Geschlecht *weiblich*. Steht in der Variablen der Wert *False*, so ist das Geschlecht *männlich*.

5. Speichern des Ergebnisses einer Division.

Das Ergebnis einer Division ist auf jedem Fall ein numerischer Wert. Allerdings ist zu beachten, dass eine Division in den wenigsten Fällen ein ganzzahliges Ergebnis liefert.

Sie sollten also für diese Übung eine Fließkommavariablen verwenden. In Frage kommen die Datentypen *Single* oder *Double*. Da keine spezielle Genauigkeit gefordert wird, sind beide folgenden Deklarationen gültige Lösungen:

```
Dim dblDivision As Single
```

oder

```
Dim dblDivision As Double
```



beschränkte Anzahl Nachkommastellen – Genauigkeit

Im Gegensatz zu einer Berechnung mit ganzen Zahlen tritt bei einer Fließkommaberechnungen immer eine Genauigkeitsabweichung auf. Der Grund hierfür ist, dass in den wenigsten Fällen bei einer Division eine endliche Anzahl Nachkommastellen als Ergebnis herauskommen.

Durch den Wertebereich ergibt sich allerdings eine beschränkte Anzahl Nachkommastellen, die in das Ergebnis übernommen werden. Der Rest wird abgeschnitten. Aus diesem abgeschnittenen Rest ergibt sich eine Ungenauigkeit. Die Stelle, an welcher der Rest abgeschnitten wird, wird als *Genauigkeit* einer Fließkommazahl bezeichnet. Je höher dieser Wert, desto höher die Genauigkeit, bzw. desto geringer die Abweichung des Ergebnisses vom tatsächlichen Ergebnis.

Die Genauigkeit des Datentyps *Single* liegt bei etwa sechs Stellen, die des Datentyps *Double* bei etwa 16 Stellen.

Unterschätzen Sie diesen Faktor nicht, wenn Sie Fließkommavariablen verwenden. Vor allem wenn Ihr Ergebnis in weiteren Berechnungen verwendet wird, können durch diese Ungenauigkeit schnell spürbare Abweichungen vom tatsächlichen Ergebnis zustande kommen.

6. Speichern Sie einen Wert bis zum nächsten Aufruf einer Funktion.

Für die Lösung dieser Übung muss das Schlüsselwort *Static* verwendet werden. Die Deklaration sieht also wie folgt aus:

```
Static intZähler as Integer
```

Da bei dieser Übung kein ablauffähiges Programm erstellt wurde, finden Sie auf der beiliegenden CD keine Programmsource.



Namenskonventionen

Es ist Ihnen vielleicht aufgefallen, dass die Variablennamen bei allen Lösungen etwas über den Inhalt und den Datentyp aussagen. Dies ist nicht zwingend notwendig. Visual Basic akzeptiert jede beliebige Zeichenkombination als Variablennamen, sofern sie den Namenskonventionen entspricht, die bereits erläutert wurden. Sie könnten also zwei Variablen in einem Programm auch folgendermaßen deklarieren:

```
Dim x1 as Integer
Dim x2 as String
```

Der Nachteil dieser »Namen« liegt auf der Hand. Falls Sie eine dieser Variablen im Programmcode sehen, können Sie anhand des Namens keinerlei Rückschlüsse auf den Verwendungszweck ziehen.

Eine weitere Möglichkeit, den Variablennamen unleserlich zu machen, sind Abkürzungen wie in den folgenden Beispielen:

```
Dim bdm as Integer
Dim vn as String
```

Sie werden zwar in diesem Fall wahrscheinlich während des Programmierens die Variablen noch erkennen, wenn Sie aber ein paar Wochen, nachdem Sie diese Zeilen geschrieben haben, das Programm erneut anpacken, wissen Sie sicher nicht mehr, dass *vn* für *Vorname* steht und *bdm* für *BetragInDm*.

Denken Sie daran, dass das Suchen der Bedeutung einer Variablen immer länger dauert als das Schreiben eines langen Variablennamens.

Aus diesen Gründen sollte der Name einer Variablen immer aussagekräftig sein und etwas über deren Bedeutung zum Ausdruck bringen. Verwenden Sie ruhig auch sehr lange Variablennamen. Wenn Sie Ihr Programm zu einem späteren Zeitpunkt wieder ändern oder erweitern, werden Sie sehr froh darüber sein.

Namenskonventionen



**aussagekräftige
Variablennamen
sparen Zeit**

3.3 Werte in Variablen speichern

Die Deklaration einer Variablen ist natürlich nur der erste Schritt, um diese in einem Programm zu verwenden. Jetzt sollten bzw. werden in der deklarierten Variablen Werte gespeichert.

Speichern eines Wertes

Das Speichern eines Wertes in einer Variablen erfolgt über eine so genannte Zuweisung. Hierzu wird ein Zuweisungsoperator benötigt. In Visual Basic ist das Gleichheitszeichen = der Zuweisungsoperator. Eine Zuweisung würde wie folgt aussehen:

```
intProzent = 16
strName = "Heinz Schwab"
```

Links vom Gleichheitszeichen steht die Variable, deren Wert gesetzt werden soll. Rechts davon steht der Wert, der zugewiesen wird. Nachdem obige Programmzeilen ausgeführt wurden, enthält die Variable *Prozent* den Wert 16 und die Variable *Name* den Wert »Heinz Schwab«.

Daten sind »variabel«

Eine einfache Zuweisung wie in den obigen Beispielen ist eher selten in einem Programm. Zumeist werden in Zuweisungen Berechnungen durchgeführt, wobei wiederum andere Variablen in der Berechnung verwendet werden. Zudem ist es (außer bei Konstanten) unüblich, einen fixen Wert einer Variablen zuzuweisen. In einem normalen Programmablauf wird der »variabel« Wert durch einen Benutzer vorgegeben oder ganz allgemein von einer Datenquelle geliefert.

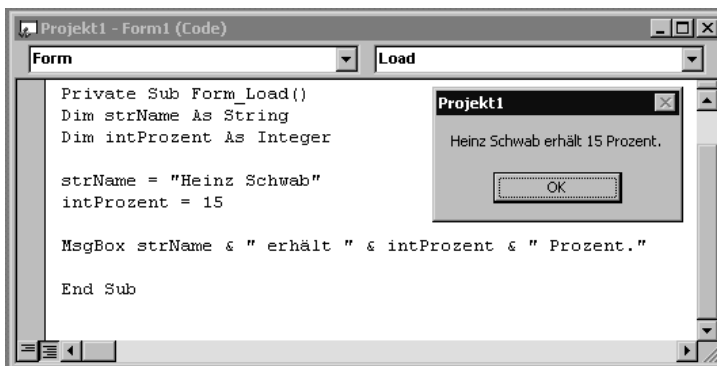
Ist in einer Variablen ein Wert gespeichert, so kann er beliebig oft in Programmanweisungen verwendet werden.

Sie könnten, beispielsweise durch Aufruf einer *MsgBox*-Anweisung, die beiden Variablen ausgeben. Die entsprechende Programmzeile würde so aussehen:

```
MsgBox strName & " erhält " & intProzent & " Prozent."
```

In dieser Anweisung werden anstatt der Variablennamen die darin gespeicherten Werte ausgegeben. Wie die Ausgabe aussieht, sehen Sie in Abbildung 3.3.

Abbildung 3.3:
Werte in Variablen
speichern und
auslesen



Die Zuweisung einer Variablen wird etwas komplizierter, wenn der Variablentyp keine Standardvariable des numerischen, bzw. des alphanumerischen Typs ist. Einer dieser Datentypen ist der Datentyp *Date*.

3.3.1 Einen Wert im Datentyp *Date* speichern

Der Datentyp *Date* wird intern als 64-Bit-Fließkommazahl gespeichert. Rein theoretisch könnten Sie die Zuweisung eines Datums also durch die Zuweisung einer Zahl erledigen. Allerdings ist die Umrechnung eines normalen Datums in die interne Darstellung etwas aufwändig. Daher wurde ein Weg gefunden, die Zuweisung in einer verständlicheren Form zu machen.

Die Zuweisung erfolgt durch Einkleiden des Datums in das Sonderzeichen `#`. Hiermit wird Visual Basic angewiesen die eingekleideten Zeichen als Datum zu interpretieren und dies in eine Fließkommazahl umzurechnen.

Um einer Datenvariablen beispielsweise Neujahr 2000, 0 Uhr 10 zuzuweisen, kann folgende Programmzeile verwendet werden:

```
dtneujahr = #1/1/2000 12:10:00 AM#
```

Der Text zwischen den Sonderzeichen `#` wird **Datumsliteral** genannt. Visual Basic akzeptiert hier mehrere gültige Formate und wandelt, falls ein interpretierbares Datum bzw. eine interpretierbare Zeit eingegeben wurde, um.

Datumsliteral

Es ist allerdings zu beachten, dass die Erkennung des Formats abhängig ist von den Ländereinstellungen Ihres Rechners. Ein Format, welches auf einem deutschen PC erkannt wird, muss auf einem englischen PC nicht unbedingt funktionieren.



3.4 Zweimal Datentyp String

In Programmen werden nicht ausschließlich numerische Werte benötigt. Auch alphanumerische Daten (Zeichenketten) müssen gespeichert werden. Für diesen Zweck gibt es den Datentyp *String*.

alphanumerische Daten

In einer Berechnung kann dieser Datentyp nicht verwendet werden. Allerdings ist es sehr wohl möglich, in einem String Zahlen zu speichern. Diese werden vor dem Speichern in eine Zeichenkette umgewandelt. Visual Basic kann diese Umwandlung zumeist automatisch vornehmen.

In Visual Basic gibt es den Datentyp *String* zweimal. Zum einen als *dynamischen String*, zum anderen als *statischen String*.

Bei einem *dynamischen String* wird bei der Deklaration keine Länge angegeben. Diese wird während des Programmlaufes automatisch dann festgelegt, wenn eine Zeichenkette zugewiesen wird. Die Länge eines dynamischen Strings entspricht immer der Länge der gespeicherten Zeichenkette.

dynamischer String

statischer String Bei einem *statischen* String wird die Länge bei der Deklaration festgelegt. Sie ist nachträglich nicht mehr veränderbar. Ein String fester Länge verändert seine Länge nicht, wenn eine Zeichenkette zugewiesen wird. Ist die zugewiesene Zeichenkette kürzer als die deklarierte Länge, so wird bis zur definierten Länge mit Leerzeichen aufgefüllt. Ist die zugewiesene Zeichenkette länger, so werden die überschüssigen Zeichen der zugewiesenen Zeichenkette abgeschnitten.

Zeichenketten werden in Visual Basic zur Kennzeichnung des Anfangs und des Endes in doppelte Anführungszeichen gesetzt. Ein Beispiel hierfür ist folgende Anweisung:

```
strName = "Max Müller"
```

Da Zeichenketten keine numerischen Variablen sind, können sie nicht in Berechnungen mittels Rechenoperatoren miteinander verknüpft werden. Trotzdem gibt es einen speziellen Operator für Zeichenketten. Es handelt sich dabei um den Operator &.

Zeichenketten verketteten

Der Operator & verbindet zwei Zeichenketten miteinander. In Visual Basic kann wahlweise auch der Operator + für diese Operation verwendet werden. Es gibt allerdings Situationen, in denen nicht klar ist, ob eine Verkettung oder eine wirkliche Addition stattfinden soll. Wenn Sie den Operator & verwenden, kann Visual Basic Sie bei der Kompilierung warnen. Zudem ist es Visual Basic möglich, eine automatische Konvertierung durchzuführen.

Folgendes Beispiel zeigt den Unterschied zwischen der Verwendung des Operators & und der Verwendung des Operators + zur Verkettung von Zeichenketten ganz deutlich.



```
Dim strTest As String  
strTest = " Bier"  
MsgBox 4 + strTest
```

In obigem Beispiel wird zunächst die Variable *strTest* als *String*-Variable deklariert. Der Variablen wird eine Zeichenkette zugewiesen. Anschließend werden durch die Anweisung *MsgBox* eine 4 und die Zeichenkette *strTest* verkettet. Es wird hierfür der Operator + verwendet.

In Abbildung 3.4 sehen Sie das Ergebnis, wenn dieses Programm in Visual Basic gestartet wird. In der Zeile mit der Anweisung *MsgBox* soll die Zahl 4 durch den Operator + mit der Zeichenkettenvariablen *strTest* verkettet und ausgegeben werden.

Da jedoch die Zahl 4 für dieses Beispiel ohne die Einkleidungszeichen für eine Zeichenkette geschrieben wurde, nimmt Visual Basic an, dass nicht ein verketteter String, sondern eine Berechnung durchgeführt werden soll. Ein String kann jedoch nicht zu einem numerischen Wert hinzuaddiert werden. Das Resultat ist ein Laufzeitfehler und somit der Abbruch des Programms.

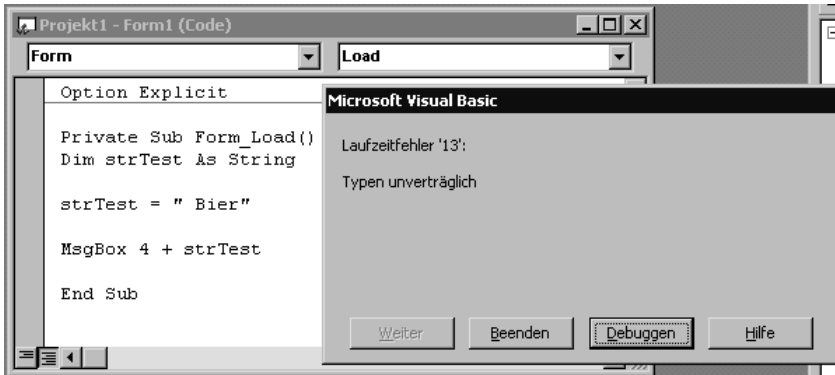


Abbildung 3.4:
Verkettung von
Zeichenketten
mit Operator +

In Abbildung 3.5 wurde der Operator + durch den speziellen String-Verkettungs-Operator & ersetzt. Jetzt tritt kein Laufzeitfehler auf. Stattdessen wird in einem Dialog die Zeichenkette *4test* ausgegeben.

Was ist passiert?

Durch die Verwendung des speziellen Verkettungs-Operators & kann Visual Basic erkennen, dass keine Berechnung durchgeführt werden soll. Daher konvertiert Visual Basic automatisch die Zahl 4 in einen String und verkettet diesen dann mit der Variablen *strTest*.

**automatische
Konvertierung**

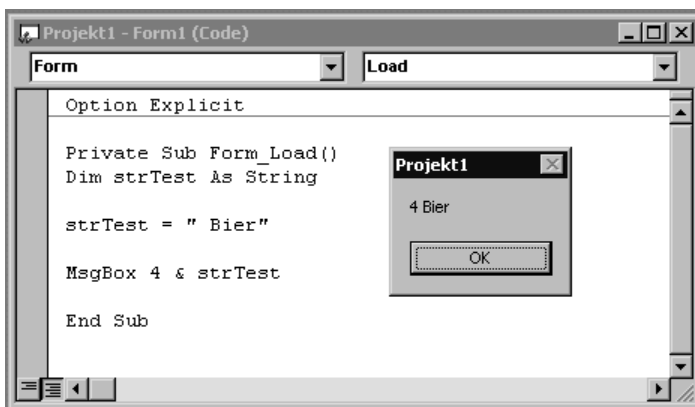


Abbildung 3.5:
Verkettung von
Zeichenketten mit
Operator &

Wenn Sie Zeichenketten verketteten, sollten Sie immer den speziellen Verkettungsoperator & verwenden. Eine Fehlinterpretation als Berechnung kann dann nicht auftreten.





3.4.1 Übung: Dynamische und statische Strings

Schreiben Sie ein kleines Programm, welches den Unterschied zwischen *dynamischen* und *statischen* Strings demonstriert.

Lösung

Folgendes Beispiel zeigt den Unterschied zwischen *dynamischen* und *statischen* Strings in Visual Basic:



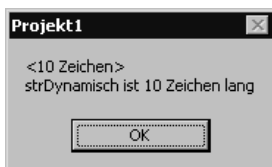
```
Dim strDynamisch As String
Dim strFix As String * 11
strDynamisch = "10 Zeichen"
strFix = "10 Zeichen"
MsgBox "<" & strDynamisch & ">" & vbCrLf & "strDynamisch ist " &
Len(strDynamisch) & " Zeichen lang"
MsgBox "<" & strFix & ">" & vbCrLf & "strFix ist " & Len(strFix) & "
Zeichen lang"
strDynamisch = "mehr als 10 Zeichen"
strFix = "mehr als 10 Zeichen"
MsgBox "<" & strDynamisch & ">" & vbCrLf & "strDynamisch ist " &
Len(strDynamisch) & " Zeichen lang"
MsgBox "<" & strFix & ">" & vbCrLf & "strFix ist " & Len(strFix) & "
Zeichen lang"
```

In den ersten zwei Zeilen des Programms werden zwei Variablen des Datentyps *String* deklariert. Die erste Deklaration definiert die Zeichenkette *strDynamisch* als *String* variabler Länge. Die zweite Deklaration definiert eine *statische* Zeichenkette mit einer festen Länge von elf Zeichen.

In den nächsten beiden Zeilen werden den Variablen Werte zugewiesen. Es handelt sich bei beiden Zuweisungen um die identische Zeichenkette mit einer Länge von 10 Zeichen.

Anschließend werden die Variablen einzeln mit der Funktion *MsgBox* ausgegeben, wobei die Länge der Variablen über die Funktion *Len* festgestellt und ebenfalls ausgegeben wird.

Abbildung 3.6:
Dynamische
Zeichenkette mit
Länge 10



In Abbildung 3.6 sehen Sie die Ausgabe der ersten *MsgBox*-Anweisung des Programms.

Sie werden feststellen, dass der ausgegebene Text zweizeilig ist. Dies wird durch die Zeichenkettenkonstante `vbCrLf` bewerkstelligt, die einen Zeilenvorschub innerhalb der auszugebenden Zeichenkette bewirkt.

Die Konstante ist in Visual Basic 6.0 vordefiniert, d.h. sie wird Ihnen von der Programmierumgebung zur Verfügung gestellt. Wenn Sie in Visual Basic die Eingabemarke auf die Konstante stellen und die Funktionstaste `[F1]` drücken, erhalten Sie weitere vordefinierte Zeichenkettenkonstanten, die Ihnen das Arbeiten mit Zeichenketten vereinfachen.

Zeilenvorschub

vordefinierte Zeichenkettenkonstanten



Abbildung 3.7:
Zeichenkette fixer
Länge nach
Zuweisung

Abbildung 3.7 zeigt die Ausgabe der *MsgBox*-Anweisung für die Zeichenkette fixer Länge. Sie werden sicher feststellen, dass die Länge der Variablen elf ist, obwohl nur zehn Zeichen zugewiesen wurden. Wenn Sie die erste Zeile genauer betrachten, werden Sie des Weiteren feststellen, dass nach dem letzten Zeichen tatsächlich noch ein elftes erscheint.

Die Erklärung hierfür ist recht einfach. Da die Zeichenkette mit einer fixen Länge von 11 Zeichen deklariert wurde, werden durch die Zuweisung zwar die ersten zehn Zeichen überschrieben, aber das elfte Zeichen bleibt dennoch vorhanden.

Das Beispiel zeigt deutlich den Unterschied zwischen Zeichenketten fixer und variabler Länge. Obwohl beiden Variablen die gleiche Zeichenkette zugewiesen wird, haben sie anschließend unterschiedliche Längen. *Dynamische* Zeichenketten haben nach der Zuweisung die Länge des aktuellen Inhalts. *Statische* Zeichenketten werden durch eine Zuweisung nicht in ihrer Länge geändert.

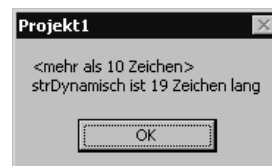


Abbildung 3.8:
Länge einer dyna-
mischen Zeichen-
ketteb nach erneu-
ter Zuweisung

Abbildung 3.8 zeigt die Ausgabe der dynamischen Variablen. Es wurde im zweiten Schritt eine längere Zeichenkette zugewiesen. Wie nicht anders zu erwarten, entspricht die Länge dem Inhalt der Variablen.

Die letzte Ausgabe des kleinen Testprogramms zeigt die Werte der Zeichenkette mit fixer Länge nach der zweiten Zuweisung. Wie Sie feststellen können, hat sich die Länge der Variablen nicht verändert. Bei der Zuweisung wurden die Zeichen, die nicht mehr in die Variable gepasst haben, abgeschnitten.

Abbildung 3.9:
Statische Zeichenkette nach
Zuweisung einer
zu langen
Zeichenkette



Werden an eine Zeichenkette mit fixer Länge mehr Zeichen zugewiesen, als gespeichert werden können, so werden die überzähligen Zeichen ohne weitere Fehlermeldung abgeschnitten. Wenn Sie in benutzerdefinierten Datentypen mit statischen Zeichenketten arbeiten, müssen Sie die Längenüberprüfung selbst programmieren.

3.5 Der multifunktionale Datentyp Variant im Detail

Variant hat »Varianten«

In einer Variant-Variablen können sowohl Zeichenketten als auch numerische Werte gespeichert werden. Er ist also ein Platzhalter für beliebige Daten. Dabei verwandelt sich der Variant bei einer Zuweisung in einen der möglichen Untertypen (Tabelle 3.2), d.h. er nimmt je nach Kontext einen passenden Datentyp an. Dies entspricht dem Verhalten eines Chamäleons, welches sich der Umgebung anpasst.

Bis auf wenige Einschränkungen können mit dem Datentyp *Variant* Rechenoperatoren oder *String*-Funktionen verwendet werden. Welche Funktionen jeweils erlaubt sind, ist abhängig vom Wert, den eine *Variant*-Variable zum Zeitpunkt der Funktionsausführung enthält. Dabei versucht der Datentyp *Variant*, falls es möglich ist, seinen Inhalt gemäß der verlangten Zuweisung anzupassen. Ist dies aufgrund des Inhalts nicht möglich, tritt ein Laufzeitfehler auf.

3.5.1 VarType

Unterdatentyp ermitteln

Eine Variant-Variablen kann zur Laufzeit eines Programms ihren Datentyp wechseln. Deshalb ist es nicht immer eindeutig, welchen Datentyp eine *Variant*-Variable, zum Zeitpunkt einer Operation, gerade innehat. Mit der *VarType*-Funktion kann er allerdings jederzeit ermittelt werden.

Der Funktion *VarType* wird eine *Variant*-Variable übergeben, der Rückgabewert besteht aus einer Zahl, die Auskunft über den Typ der Variablen gibt. Die Auswertung des *VarType*-Ergebnisses kann anhand der Tabelle 3.2 erfolgen.

Syntax `VarType(Variant-Variablen)`

Variant-Variablen können alle in Tabelle 3.2 bezeichneten Untertypen annehmen. Lediglich Strings fester Länge und benutzerdefinierte Datentypen können in einer Variant-Variablen nicht gespeichert werden.

Wert	Bedeutung	Konstante
0	Nicht initialisiert	vbEmpty
1	Null (ungültige Daten)	vbNull
2	Integer	vbInteger
3	Long Integer	vbLong
4	Single	vbSingle
5	Double	vbDouble
6	Currency	vbCurrency
7	Date	vbDate
8	String	vbString
9	Objekt	vbObject
10	Fehler	vbError
11	Boolean	vbBoolean
12	Variant	vbVariant
13	Datenzugriffsobjekt	vbDataObject
14	Decimal	vbDecimal
17	Byte	vbByte
8192	Datenfeld	vbArray

*Tabelle 3.2:
Die Untertypen des
Datentyps Variant*

Der Untertyp *Empty* bezeichnet eine Variant-Variable, der noch kein Wert zugewiesen wurde. Wird eine Variant-Variable mit diesem Inhalt in einer Operation verwendet, so nimmt sie automatisch einen Wert an, der zu der Zuweisung passt.

Untertyp *Empty*

Der Datentyp *Variant* gestaltet die Handhabung von Variablentypen sehr einfach. Solange Sie keine Strings fester Länge oder benutzerdefinierte Datentypen verwenden, werden Sie keine speziellen Datentypen benötigen. Vor allem für Programmieranfänger scheint dies geeignet zu sein, da sie sich nicht mit den verschiedenen Datentypen auseinander setzen müssen, die Visual Basic zur Verfügung stellt.

Gegen die Verwendung des Datentyps *Variant* spricht dessen Speicherverbrauch, der je Variable intern mindestens 16 Byte in Anspruch nimmt. Damit verbunden ist eine langsamere Rechengeschwindigkeit. Vor allem in größeren Programmen und aufwändigen Berechnungen ist dies deutlich spürbar.



3.5.2 Übung: Verwendung des Datentyps Variant

1. Schreiben Sie eine Funktion, die den Unterschied einer Variant-Variablen mit Wert *Empty* im Kontext als Zahl oder als String demonstriert.
2. Verwenden Sie die *Vartype*-Funktion um den Untertyp zweier unterschiedlicher Variant-Variablen festzustellen.



Lösung

Um Teil eins der Übung zu lösen, muss zunächst eine Variant-Variable mit dem Inhalt *Empty* generiert werden. Der einfachste Weg hierzu ist, eine Variant-Variable zu deklarieren und anschließend keinen Wert zuzuweisen.

Die entsprechende Anweisung könnte wie folgt aussehen:

```
Dim varTest As Variant
```

Nun muss Visual Basic dazu veranlasst werden, aus dem Programmkontext den gewünschten Variablentyp als numerisch bzw. als String festzulegen.

Für den Fall *String* kann die Variable einfach mit einem anderen String durch den Operator *&* verkettet werden. Somit weiß Visual Basic, dass diese Variable eine String-Variable sein soll:

```
"Als String <" & varTest & ">"
```



Damit Visual Basic die Variant-Variable als numerischen Wert interpretiert, muss eine Berechnung mit ihm durchgeführt werden. Um den aktuellen Inhalt der Variablen durch die Berechnung nicht zu verändern, wird eine Multiplikation mit 1 durchgeführt:

```
(1 * varTest)
```

Damit auch ein Ergebnis sichtbar wird, werden die obigen Operationen innerhalb der Funktion *MsgBox* ausgeführt. Somit wird das Ergebnis direkt in einem Bildschirmdialog veranschaulicht. Das komplette Testprogramm hat folgendes Aussehen:



```
Dim varTest As Variant  
MsgBox "Empty Variant im Kontext als Zahl 0  
& (1 * varTest) & vbCrLf &  
"Als String <" & varTest & ">"
```

Abbildung 3.10 zeigt die Ausgabe des Programmbeispiels.

Abbildung 3.10:
Datentyp Variant,
Untertyp Empty als
Zahl oder String



Bei der Ausgabe wird durch die Klammer *(1 * varTest)* eine numerische Operation durchgeführt, somit wird erzwungen, dass die Variable *varTest* als Zahl verwendet wird. Die entsprechende Ausgabe in Abbildung 3.10 zeigt, dass als Wert automatisch die 0 angenommen wird.

In der zweiten Zeile des Ausgabedialogs wird die Variable *varTest* als Zeichenkette konvertiert. In diesem Fall wird für die Variable auch folgerichtig ein Leerstring ausgegeben. Sichtbar wird es allerdings erst dadurch, dass die Variable

bei der Ausgabe zwischen spitze Klammern gesetzt wird und zwischen den spitzen Klammern kein Zeichen erscheint.

Um die zweite Teilübung zu lösen, werden zunächst zwei Variant-Variablen deklariert.

```
Dim varTestZahl As Variant  
Dim varTestString As Variant
```

Im Beispielprogramm wird der Variablen *varTestZahl* ein numerischer Wert, nämlich 5 zugewiesen. Der Variablen *varTestString* wird eine Zeichenkette, nämlich »10« zugewiesen.

```
varTestZahl = 5  
varTestString = "10"
```

Jetzt wird mit der Funktion-*MsgBox* ein Dialog (Abbildung 3.11) aufgerufen der zeigt, welchen Untertyp die Funktion *VarType* jeweils ermittelt. Die entsprechende Programmzeile sieht aus wie folgt:

```
MsgBox VarType(varTestZahl) & vbCrLf & VarType(varTestString)
```



Abbildung 3.11:
Verwendung der
VarType-Funktion

Für die Variable *varTestZahl* ermittelt die *VarType*-Funktion den Untertyp 2, was laut Tabelle 3.2 dem Untertyp *Integer*, also einem numerischen Datentyp, entspricht. Für die Variable *varTestString* ermittelt die *VarType*-Funktion den Untertyp 8. Es handelt sich also um den Untertyp *String*.

Durch die folgende Anweisung wird demonstriert, dass der Datentyp Variant ohne erneute Zuweisung seinen Untertyp gemäß des Kontextes, in dem er verwendet wird, automatisch ändert:

```
MsgBox varTestZahl * varTestString
```

Abbildung 3.12 zeigt das Ergebnis einer Multiplikation beider Variablen. Es wird als Ergebnis 50 (= 5 * 10) angezeigt. Visual Basic hat also aus dem Kontext der Programmanweisung ermittelt, dass die Variable *varTestString* als numerische Variable verwendet wird, und daher eine automatische Konvertierung vorgenommen.

Funktioniert die automatische Konvertierung immer? Um dies zu ermitteln, wird die Variable *varTestString* erneut mit einer Zeichenkette initialisiert. Diesmal werden allerdings die Buchstaben »xyz« zugewiesen:

Abbildung 3.12:
Automatische
Konvertierung
einer Variant-
Variablen

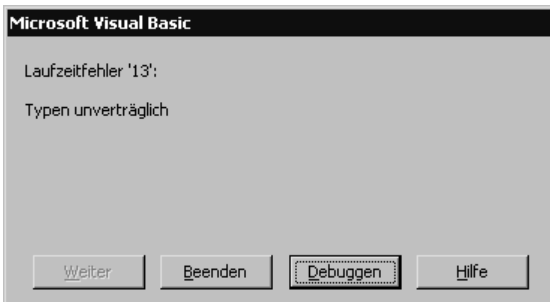


```
varTestString = "xyz"  
MsgBox varTestZahl * varTestString
```

Laufzeitfehler

Jetzt wird die Multiplikation erneut durchgeführt. Auch hier versucht Visual Basic die Variable `varTestString` automatisch zu konvertieren. Da dies jetzt jedoch nicht mehr funktioniert, weil »xyz« nun einmal nicht in eine numerische Variable konvertiert werden kann, tritt ein Laufzeitfehler auf. Sie sehen den entsprechenden Dialog in Abbildung 3.13.

Abbildung 3.13:
Die Grenzen der
automatischen
Konvertierung



Ein Laufzeitfehler beendet Ihr Programm. Läuft das Programm im Interpreter der Entwicklungsumgebung, so haben Sie die Möglichkeit über die Schaltfläche `DEBUGGEN` zu der Programmzeile zu springen, die den Fehler verursacht hat. Wurde das Programm allerdings übersetzt und läuft eigenständig, so wird eine Windows-Fehlermeldung angezeigt und das Programm ohne weitere Eingriffsmöglichkeit beendet.



Bei einer Variant-Variablen besteht immer die Gefahr, dass ein Wert zugewiesen wird, der nicht dem erforderlichen Datentyp entspricht. Ein spezieller Datentyp wird eine solche Zuweisung verhindern. Damit ist gewährleistet, dass bei einer späteren Verwendung der Variablen der Datentyp immer stimmt.

3.6 Benutzerdefinierte Datentypen

eine Ansamm- lung/Zusammen- fassung

Benutzerdefinierte Datentypen werden vom Programmierer selbst definiert. Es handelt sich dabei streng genommen nicht um einen wirklichen Datentyp, sondern vielmehr um eine Ansammlung/Zusammenfassung logisch zusammengehöriger Variablen unter einen Namen.

Innerhalb eines benutzerdefinierten Datentyps werden nur Datentypen verwendet, die Visual Basic kennt. Wenn es für eine übersichtliche Strukturierung von Vorteil ist, können auch benutzerdefinierte Datentypen als Elemente eines benutzerdefinierten Datentyps verwendet werden. Diese müssen dann allerdings im Programm in der richtigen Reihenfolge definiert werden.

**übersichtliche
Strukturierung**

Mit Hilfe der *Type*-Anweisung wird die Struktur bzw. Zusammensetzung des benutzerdefinierten Datentyps bekannt gemacht. Danach können Variablen dieses Typs deklariert werden.

```
[Private | Public] Type VarName
    Element As Datentyp
    Element As Datentyp
    ...
    ...
End Type
```

Syntax

Nach der Definition des benutzerdefinierten Datentyps kann eine Variable dieses Datentyps im Programm deklariert werden. Hierzu wird eine Deklarationsanweisung im Programmcode eingefügt, die als Variablentyp den Namen (*Var-Name*) der Datentypdefinition verwendet. Folgendes Beispiel zeigt die Funktionsweise:

```
Private Type Auto
    Marke As String
    Baujahr As Date
    Leistung As Integer
    Kilometerstand As Long
End Type
Dim udtMeinAuto As Auto
```



Im Datentyp *Auto* können keine Werte gespeichert werden. Er ist nur die Definition, die Schablone, für den Datentyp. Erst wenn mit der Anweisung *Dim* eine Variable dieses Typs angelegt wird, können auch Werte gespeichert werden.

Die Zuweisung von Werten in eine benutzerdefinierte Variable erfolgt über Einzelzuweisungen der Teilvariablen. Dabei wird die Teilvariable durch einen Punkt vom Variablennamen getrennt. Folgende Programmzeilen zeigen eine solche Zuweisung:

```
udtMeinAuto.Kilometerstand = 300000
```

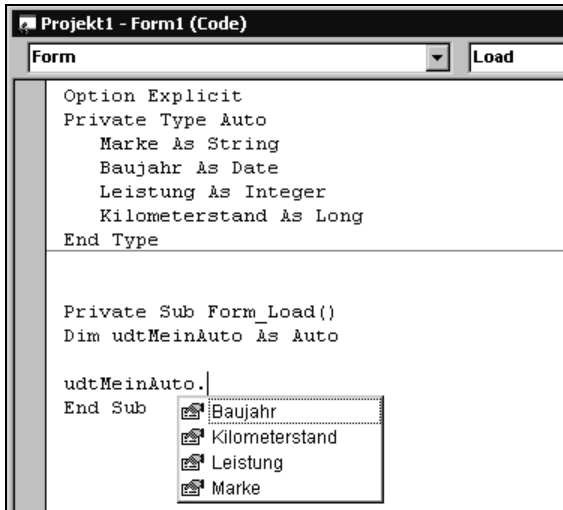
**Teilvariablen
zuweisen**

Visual Basic unterstützt Sie dabei mit der automatischen Befehlsergänzung des Codefensters. Die automatische Befehlsergänzung (Abbildung 3.14) wird angezeigt, sobald Sie nach dem Namen der benutzerdefinierten Variable einen Punkt schreiben.

Bei einer Zuweisung darf die benutzerdefinierte Variable selbst nur dann verwendet werden, wenn eine komplette andere Variable des gleichen benutzerdefinierten Datentyps zugewiesen wird.

**komplette
Variable zuweisen**

Abbildung 3.14:
Hilfe bei der
Eingabe der
Teilvariablen



```
Dim udtMeinAuto As Auto
Dim udtDeinAuto As Auto
udtMeinAuto.Marke = "Rostlaube"
udtDeinAuto = udtMeinAuto
MsgBox udtDeinAuto.Marke
```

Im Beispiel werden zwei Variablen des benutzerdefinierten Datentyps *Auto* deklariert. Der Variablen *udtMeinAuto.Marke* wird ein Wert zugewiesen. Anschließend wird die komplette Variable *udtMeinAuto* der noch nicht initialisierten Variable *udtDeinAuto* zugewiesen.

Die anschließende Ausgabe von *udtDeinAuto.Marke* (Abbildung 3.15) zeigt, dass bei der Zuweisung der beiden benutzerdefinierten Variablen der Inhalt von *udtMeinAuto* nach *udtDeinAuto* kopiert wurde.

Abbildung 3.15:
Ausgabe nach einer
Zuweisung



Einen zwingenden Grund für den Einsatz von benutzerdefinierten Datentypen gibt es nicht. In manchen Situationen ist es von Vorteil, Daten zu gruppieren, um einen logischen Zusammenhang im Programm zu dokumentieren.

Direktzugriffsdateien

Häufig werden benutzerdefinierte Variablen im Zusammenhang mit Datenfeldern und Direktzugriffsdateien verwendet. Bei der Verwendung in Datenfeldern ergibt sich der Vorteil, dass in *einem* Feld mehrere Variablen unterschiedlichen Datentyps unter *einem* Schlüssel gespeichert werden können.

Bei der Verwendung mit Direktzugriffsdateien muss nicht jede Variable einzeln eingelesen oder geschrieben werden, sondern es kann durch eine *Get-* oder *Put-*Anweisung immer ein kompletter »Datensatz«, nämlich der benutzerdefinierte Typ, gelesen oder geschrieben werden.

3.6.1 Übung: Verwendung benutzerdefinierter Datentypen



1. Erstellen Sie einen benutzerdefinierten Datentyp, der in der Lage ist, die Vornamen und Namen und Adressdaten einer Person zu speichern.
2. Deklarieren Sie ein Variable dieses Typs und füllen Sie diese Variable mit Werten.
3. Geben Sie die gespeicherten Werte der Variablen in einem Dialog aus.

Lösung

Teilübung eins beschränkt sich darauf, einen benutzerdefinierten Datentyp zu generieren.

Die entsprechende Deklaration sieht aus wie folgt:

```
Type Person
  Vorname As String
  Name As String
  Geburtsdatum As Date
  Telefonnummer As String
  Postleitzahl As Long
  Wohnort As String
  Strasse As String
  Hausnummer As Integer
End Type
```

Bei der Definition eines benutzerdefinierten Datentyps sollten Sie bereits wissen, für welchen Zweck er in Ihrem Programm verwendet wird. Sie könnten beispielsweise die Adresse auch in einer einzigen Variablen ablegen, wobei die Postleitzahl und der Wohnort in einer Zeichenkette gespeichert werden.

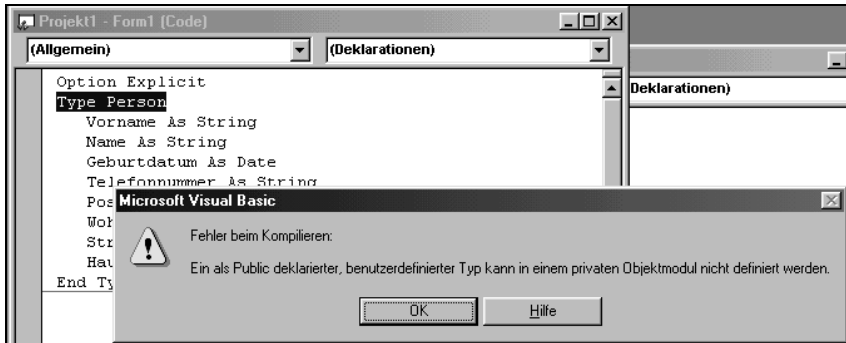
Allerdings würde in diesem Fall die Suche einer Adresse in einem bestimmten Bereich schwierig ausfallen. Ist die Postleitzahl einzeln aufgeführt und der Datentyp eine numerische Variable, so können sehr einfach Vergleiche programmiert werden.

Falls Sie nach der Definition eines benutzerdefinierten Datentyps beim Start des Programms die Fehlermeldung aus Abbildung 3.16 zu sehen bekommen, wurde der Datentyp am falschen Ort deklariert.



In diesem Fall müssen Sie Ihrem Projekt ein Standardmodul hinzufügen und die Deklaration des benutzerdefinierten Datentyps in den allgemeinen Teil dieses Moduls verschieben.

Abbildung 3.16:
Definition benutzerdefinierter
Datentypen nur in
öffentlichen
Modulen



Die Deklaration einer Variablen und die Zuweisung von Daten für die zweite Teilübung wird mit folgenden Programmzeilen erledigt:

```
Dim einePerson As Person
einePerson.Vorname = "Captain"
einePerson.Name = "Future"
einePerson.Geburtsdatum = #1/1/2030#
einePerson.Telefonnummer = "99999/12345"
einePerson.Postleitzahl = 99999
einePerson.Wohnort = "Mondstadt"
einePerson.Strasse = "Meteoritenweg"
einePerson.Hausnummer = 777
```

Bei der Deklaration wird nach dem Schlüsselwort *As* als Datentyp der neue benutzerdefinierte Datentyp *Person* angegeben. Der Datentyp selbst kann für Zuweisungen nicht verwendet werden. Erst die deklarierte Variable *einePerson* steht hierfür zur Verfügung.

Bei den anschließenden Zuweisungen werden die einzelnen Elemente des Datentyps durch den Punkt getrennt vom Variablennamen spezifiziert. Zugeewiesen wird jeweils ein Wert, der zum Datentyp des Elements passt. Bei dem Element *Vorname* beispielsweise eine Zeichenkette, bei *Hausnummer* ein numerischer Wert und bei *Geburtsdatum* ein Datumsliteral.

Für die dritte Teilübung wird eine zusätzliche Variable generiert. Sie wird vor der Ausgabe mit dem auszugebenden Text gefüllt. Dieses Vorgehen hat den Vorteil, dass der Programmcode etwas übersichtlicher wird, denn sonst müsste die komplette auszugebende Zeichenkette in einer Zeile stehen.

Daher sind folgende Programmzeilen geschrieben worden:

```
Dim message As String
message = "Für " & einePerson.Vorname & " " & einePerson.Name & "
wurden folgende Daten eingegeben" & vbCrLf
message = message & "Geboren am " & einePerson.Geburtsdatum & vbCrLf
message = message & "Telefonisch erreichbar unter " &
einePerson.Telefonnummer & vbCrLf
```

**Übersichtlicher
durch Zusatz-
variable**

```

message = message & " Adresse ist: " & vbCrLf
message = message & einePerson.Strasse & " " & einePerson.Hausnummer &
vbCrLf
message = message & einePerson.Postleitzahl & " " & einePerson.Wohnort
& vbCrLf
MsgBox message

```

Wenn Sie das Programm starten, erhalten Sie den Dialog aus Abbildung 3.17.

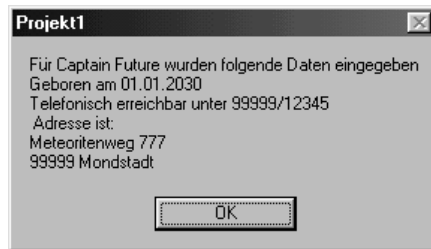


Abbildung 3.17:
Ausgabe der Daten
des benutzerdefinierten
Datentyps

3.7 Feldvariablen

Eine spezielle Art der Variablendeklaration ergibt sich durch die Verwendung einer Feldgröße bei der Variablendeklaration. Hierdurch wird eine so genannte Feldvariable angelegt. In einer Feldvariablen werden Variablen gleichen Datentyps gespeichert. Für eine Feldvariable wird auch oft der englische Begriff *Array* verwendet.

Array

Eine Feldvariable bietet dann einen entscheidenden Vorteil, wenn Berechnungen oder sonstige Aktionen mit vielen gleichartigen Variablen durchgeführt werden müssen.

Jede einzelne Variable eines Feldes wird über den so genannten *Index* angesprochen. Der Index entspricht der Position einer speziellen Variablen innerhalb eines Feldes.

```

Dim Feldvariablenname [( [ [Untergrenze To] Obergrenze] [,
[Untergrenze To] Obergrenze] ... )] [As [New] Datentyp]

```

Syntax

Durch die Angabe mehrerer Grenzwerte wird eine Feldvariable mehrdimensional. Um in einem mehrdimensionalen Feld eine spezielle Variable zu adressieren, müssen mehrere Indexwerte angegeben werden.

mehrdimensionale Felder

Wird bei der Angabe einer Dimension die Untergrenze weggelassen, so wird die Untergrenze automatisch mit 0 vorbelegt. Folgende Definitionen sind also identisch:

```

Dim intArray (5) as Integer
Dim intArray (0 to 5) as Integer

```

3.7.1 Übung: Programmieren mit einer Feldvariablen



1. Deklarieren Sie eine Feldvariable mit Unter- und Obergrenze und geben Sie die erste und letzte Variable des Feldes aus.



2. Deklarieren Sie eine Feldvariable ohne Untergrenze und geben Sie die erste und letzte Variable aus.

Lösung

Um Teil eins der Übung zu lösen, wird zunächst eine Feldvariable über folgende Programmzeile deklariert:

```
Dim arrInteger(1 To 5) As Integer
```

Die Übung gibt den Variablentyp und den Bereich nicht vor, es wurden daher der Datentyp *Integer* und ein Bereich von 1 bis 5 als Feldgrenzen willkürlich gewählt. Jede andere Lösung wäre natürlich auch richtig.

Mit dieser Deklaration wird ein Feld angelegt, welches insgesamt fünf Elemente beinhaltet. Um die einzelnen Elemente bzw. Variablen des Feldes anzusprechen, wird der Variablenname gefolgt vom Index in Klammern verwendet.

```
arrInteger(1) = 1  
arrInteger(5) = 5
```

Durch obige Programmzeilen wird in der Variable auf Index 1 der Wert 1 gespeichert, auf Index 5, der Obergrenze des Feldes, wird der Wert 5 gespeichert.

Die Ausgabe der Variablen kann durch eine MsgBox-Methode geschehen:

```
MsgBox "Untergrenze " & arrInteger(1) & vbCrLf & "Obergrenze " &  
arrInteger(5)
```

Die Ausgabe dieser MsgBox sehen Sie in Abbildung 3.18.

Abbildung 3.18:
Ausgabe zweier
Feldvariablen



Für die zweite Teilübung wird ein Feld ohne Untergrenze deklariert. Es wird folgende Programmzeile verwendet:

```
Dim arrInteger(5) as Integer
```


Um die erste und die letzte Variable des Feldes auszugeben, muss man wissen, dass Visual Basic als Untergrenze automatisch den Wert 0 annimmt, wenn dieser in der Deklaration weggelassen wird. Das oben deklarierte Feld hat also im Gegensatz zum Feld der ersten Teilübung insgesamt sechs Elemente, da die Untergrenze 0 ist.



Daraus ergeben sich auch weitere Änderungen im Programm, die in folgenden Programmzeilen zu sehen sind:

```
arrInteger(0) = 0
arrInteger(5) = 5
MsgBox "Untergrenze " & arrInteger(0) & vbCrLf & "Obergrenze " &
arrInteger(5)
```

Sowohl in der Zuweisung als auch bei der Ausgabe der Variablen wird jetzt für das erste Element des Feldes auf den Index 0 zugegriffen.

In Teil zwei der Übung wurde die Untergrenze des Feldes nicht direkt angegeben bzw. in der Deklaration bestimmt. Oft wird für den Index auch eine Variable verwendet. In diesem Fall könnte es vorkommen, dass Sie versuchen auf einen Index zuzugreifen, der nicht existiert.



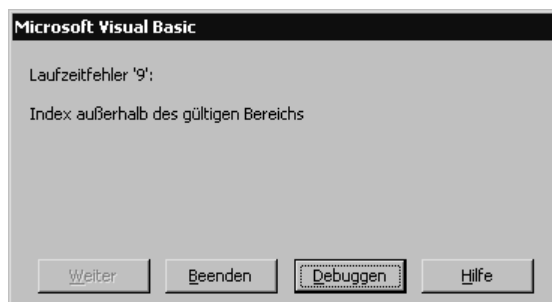
Was passiert in diesem Fall?

Um dies zu testen, können Sie in einem der beiden obigen Programme die Zeile mit der *MsgBox*-Methode wie folgt ändern:

```
MsgBox "Untergrenze " & arrInteger(0) & vbCrLf & "Obergrenze " &
arrInteger(6)
```

Da der Index 6 im Feld nicht vorhanden ist, tritt bei der Abarbeitung der Programmzeilen der Laufzeitfehler 9 (Abbildung 3.19) auf. Dieser Fehler ist eigens für eine Indexüber- oder -unterschreitung generiert worden. Sie können daran vielleicht erkennen, dass dieser Fehler öfter auftritt, wenn mit Feldvariablen programmiert wird.

**Indexüber- oder
-unterschreitung**



*Abbildung 3.19:
Index außerhalb
des gültigen
Bereichs*

Laufzeitfehler 9 hat, wie jeder andere Laufzeitfehler, die unangenehme Eigenschaft, Ihr Programm zu beenden, falls Sie nicht geeignete Maßnahmen wie ein *Error-Handling* programmieren.



Testen Sie bei der Programmierung mit Feldern immer sehr genau, ob die Indexgrenzen Ihres Feldes unter allen Umständen beachtet werden.

3.7.2 Unter- und Obergrenze einer Feldvariablen ermitteln

Wenn in einem Programm die Indexgrenzen eines Feldes nicht bekannt sind bzw. sich dynamisch ändern, kann es notwendig sein, diese zu ermitteln.

Zu diesem Zweck existieren die Funktionen *LBound* bzw. *UBound*.

Bound, die Grenze

Die Buchstaben vor dem *Bound* (engl. Grenze) weisen auf den jeweiligen Zweck hin. L steht für *Lower*, also die untere Grenze, und U steht für *Upper*, die obere Grenze.

Syntax

```
UBound(Feldvariablenname)
```

```
LBound(Feldvariablenname)
```

Beide Funktionen liefern als Rückgabewert eine Variable des Datentyps Long.



3.7.3 Übung: Grenzen von Feldern ermitteln

Deklarieren Sie ein Feld mit zehn Elementen. Die Obergrenze des Feldes ist 5. Geben Sie die Obergrenze und die Untergrenze des Feldes in einem Dialog aus. Verwenden Sie hierzu die Funktionen *Ubound* und *Lbound*.

Lösung

Um die Feldvariable gemäß der Übung zu deklarieren, muss als Untergrenze ein negativer Wert angegeben werden. Beachten Sie bitte, dass dabei der Index 0 mit eingeschlossen wird. Um zehn Elemente zu deklarieren, muss die Untergrenze des Feldes also -4 sein.

In folgender Programmzeile wird die korrekte Deklaration durchgeführt:

```
Dim arrInteger(-4 To 5) As Integer
```

Um die Grenzen des Feldes auszugeben, wird die bewährte *MsgBox*-Funktion verwendet. Die Ermittlung der Grenzen und deren Ausgabe erfolgt in einer Programmzeile:

```
MsgBox "Untergrenze " & LBound(arrInteger) & vbCrLf & "Obergrenze " & UBound(arrInteger)
```

In Abbildung 3.20 sehen Sie die Ausgabe dieser Programmzeile.



Abbildung 3.20:
Unter- und Ober-
grenze einer
Feldvariablen

Wie im Beispiel gezeigt, muss eine Feldvariable nicht bei 0 oder 1 beginnen. Es ist durchaus möglich, für die beiden Grenzwerte auch negative Zahlen zu verwenden. Allerdings muss darauf geachtet werden, dass die Untergrenze kleiner ist als die Obergrenze. Durch die Deklaration eines zweidimensionalen Feldes haben Sie so beispielsweise die Möglichkeit ein Koordinatensystem mit x- und y-Achse abzubilden.



3.8 Konstanten

Der Inhalt einer Variablen kann sich während der Laufzeit einer Anwendung jederzeit durch verschiedene Verarbeitungen verändern. Im Gegensatz hierzu repräsentiert eine Konstante einen festen Wert, der zur Laufzeit durch keine Anweisung modifiziert werden kann.

feste Werte

3.8.1 Literale

Visual Basic unterscheidet zwei Arten von Konstanten. Bei der einen Art handelt es sich um beliebige Zahlen oder Zeichenketten, die direkt im Programmcode angegeben werden. Diese Konstanten werden als *Literale* bezeichnet.

Die Zuweisungen in den bisherigen Übungen wurden zumeist mit *Literalen* gemacht.

In den folgenden Programmzeilen ist die Zahl 10, der Name »Heinz Schwab« und das Datum ein *Literal*. Sie alle werden im Codefenster eingegeben und können während der Laufzeit nicht mehr verändert werden.

```
intMengeInteger = 10
udtMeinAuto.Marke = "Schrottlaupe"
dtDatum = #1/2/2000 2:05:06 AM#
```

Bei *Zeichenkettenliteralen* werden der Anfang und das Ende des Literals durch doppelte Hochkommas gekennzeichnet. *Datumsliterale* werden in zwei Doppelpunkte eingeschlossen.

3.8.2 Symbolische Konstanten

Die zweite Art der Konstante wird als benannte oder symbolische Konstante bezeichnet. Sie wird mit Hilfe des Schlüsselwortes *Const* definiert.

**benannte oder
symbolische
Konstante**

Syntax Const KonstName [As Typ] = Ausdruck

Der Parameter *KonstName* bezeichnet den Namen der Konstanten. Unter diesem Namen kann der zugewiesene Ausdruck später im Programm verwendet werden.

Damit normale Variablen und Konstanten sich im Programmcode sichtbar unterscheiden, sollten Sie symbolische Konstanten immer in Großbuchstaben schreiben.

Die Festlegung des Datentyps der Konstante über *As Typ* ist optional. Wird sie weggelassen, verwendet Visual Basic den bereits bekannten Datentyp *VARIANT*.

Mit der Zuweisung = *Ausdruck* erhält die Konstante ihren Wert. Dieser Wert kann während der Laufzeit des Programms nicht mehr geändert werden.

lesbarer, einfacher, schneller

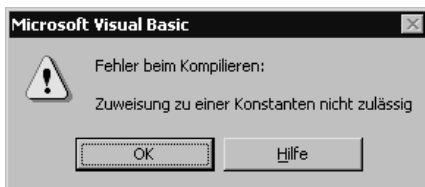
Symbolische Konstanten werden immer dann verwendet, wenn ein fest definierter Wert in einer Anwendung mehrmals benötigt wird. Der Programmcode wird nicht nur wesentlich lesbarer, es ist auch viel einfacher und schneller möglich, ihn an neue Gegebenheiten anzupassen. Denn der Wert muss nur einmal, nämlich bei der Konstantendefinition, geändert werden.

```
Const DATEN_VERZEICHNIS As String = "C:\GEHEIM"
```

```
Const ZEILEN_PRO_SEITE As Integer = 60
```

Die Verwendung symbolischer Konstanten vermeidet auch das versehentliche Überschreiben eines Wertes. Wird im Programm einer Konstanten ein Wert zugewiesen, so reagiert Visual Basic mit der Fehlermeldung *Zuweisung zu einer Konstanten nicht zulässig* (Abbildung 3.21).

Abbildung 3.21:
Konstanten können
im Programm nicht
verändert werden



kein Laufzeitfehler!

Es handelt sich bei dieser Fehlermeldung nicht um einen Laufzeitfehler. Visual Basic weigert sich bereits das Programm zu übersetzen bzw. im Interpreter laufen zu lassen. Zudem werden Sie nach Betätigen der Schaltfläche *OK* an genau der Stelle im Programmcode landen, welche den Fehler verursacht hat.

3.8.3 Vordefinierte Konstanten

Visual Basic verwendet den Mechanismus der Konstanten selbst. Für häufig verwendete Werte hat Visual Basic bereits eine Vielzahl von Konstanten definiert. Diese werden als *vordefinierte Konstanten* bezeichnet. Sie stehen jedem Visual Basic-Programm ohne zusätzliche Definitionen zur Verfügung.

In einigen Beispielen dieses Kapitels wurde bereits die vordefinierte Zeichenkettenkonstante `vbCrLf` verwendet, die einen Zeilenumbruch veranlasst, wenn sie in einen String eingefügt wird.

Eine Übersicht über alle Visual Basic-Konstanten gibt der *Objektkatalog*, der in der Entwicklungsumgebung mit der Taste F2 aufgerufen werden kann.

Objektkatalog

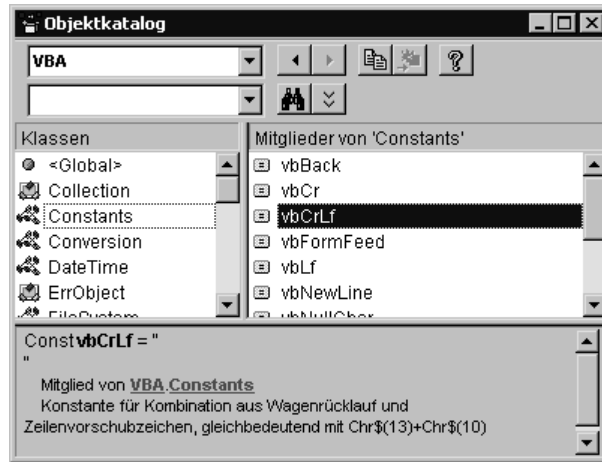


Abbildung 3.22:
Vordefinierte
Konstanten im
Objektkatalog

In Abbildung 3.22 wurde durch Auswahl des Eintrags *VBA* in der *Projekt/Bibliotheks*-Liste und durch die anschließende Auswahl des Eintrags *Constants* in der *Klassen*-Liste die Auswahl der vordefinierten Konstanten angezeigt.

Wird in der *Mitglieder*-Liste eine spezielle Konstante ausgewählt, so zeigt der untere Bereich des Fensters eine kurze Erklärung und den Wert der Konstanten an.

3.9 Gültigkeitsbereiche von Variablen und Konstanten

Der Gültigkeitsbereich einer Variablen entscheidet darüber, wer sie verwenden kann bzw. von wo aus auf sie zugegriffen werden darf. So ist es durchaus möglich, dass Sie in einem Programm eine Zählvariable gleichen Namens, z. B. `i`, an mehreren Stellen deklarieren und es sich trotzdem – obwohl diese Variablen denselben Namen haben – um verschiedene Variablen handelt, da sie in unterschiedlichen Gültigkeitsbereichen wirksam sind.

Über den Gültigkeitsbereich entscheidet zum einen das Deklarationsschlüsselwort, zum anderen der Ort der Deklaration.

Gültigkeitsbereich

Achten Sie darauf, dass Sie nicht Variablen gleichen Namens in sich überschneidenden Gültigkeitsbereichen deklarieren.



3.9.1 Gültigkeitsbereich von Variablen

In Visual Basic können Variablen in drei verschiedene Gültigkeitsbereiche eingeordnet werden:

- ▶ Lokale Variablen
- ▶ Private Variablen
- ▶ Öffentliche Variablen

Die lokale Variable

Wird eine Variable innerhalb einer Prozedur oder Funktion deklariert, so ist sie auch nur dort verwendbar. Diese Variable hat einen lokalen, oder besser Prozedur-lokalen, Gültigkeitsbereich.

**nur aktiv, wenn
Funktion aktiv**

Lokale Variablen werden bei jedem Aufruf einer Funktion neu angelegt und initialisiert. Ist die Funktion nicht aktiv, so existieren auch die lokalen Variablen nicht.

Die private Variable

Als privat wird eine Variable bezeichnet, wenn sie einen Gültigkeitsbereich hat, der sich über das ganze Modul, in welcher sie deklariert wurde, erstreckt. Allerdings ist eine private Variable nicht in anderen Modulen eines Projekts verfügbar.

Damit eine Variable im kompletten Modul verfügbar ist, muss sie im Allgemein- oder Deklarationsteil des Moduls deklariert werden.

**innerhalb eines
Moduls sichtbar**

Existiert innerhalb einer Funktion des Moduls eine lokale Variable gleichen Namens, so wird innerhalb dieser Prozedur nicht die private, auf Modulebene deklarierte Variable, sondern die interne, lokale Variable angesprochen.



Folgendes Beispiel zeigt dieses Verhalten:

```
Private intAlter As Integer
Private Sub Command1_Click()
    Dim intAlter As Integer
    MsgBox intAlter
End Sub
Private Sub Form_Load()
    intAlter = 15
End Sub
```

Im allgemeinen Bereich eines Form-Moduls wird die *Private*-Variable mit dem Namen *intAlter* deklariert. Auf diese Variable kann im ganzen Modul zugegriffen werden. Dies wird deutlich, da ihr in der Ereignisprozedur *Form_Load* der Wert 15 ohne lokale Deklaration zugewiesen werden kann.

In der Ereignisprozedur *Command1_Click* wird eine weitere Variable mit dem Namen *intAlter* lokal deklariert. Diese wird ohne weitere Zuweisung in die *MsgBox* aus Abbildung 3.23 ausgegeben.



Abbildung 3.23:
Lokale Variable hat Vorrang

Der Wert der Variablen *intAlter* ist nicht 15, sondern 0! Da die im allgemeinen Teil des Moduls deklarierte *Private*-Variable mit Sicherheit nach der Zuweisung in der *Form_Load*-Ereignisroutine nicht mehr verändert wurde, muss es sich hierbei um die lokal deklarierte Variable gleichen Namens handeln.

lokal geht vor global

Die lokal deklarierte Variable hat deshalb den Wert 0, weil sie beim Aufruf der Funktion neu angelegt und initialisiert wurde.

Das Schlüsselwort *Private* darf nicht auf Prozedurebene verwendet werden, da hier ausschließlich lokale Variablen erlaubt sind, die über die *Dim*-Anweisung deklariert werden müssen. Eine Variable, die mit dem Schlüsselwort *Private* deklariert wurde, kann hingegen im Allgemeinteil eines Moduls auch über eine *Dim*-Anweisung deklariert werden.

Wird eine *private* Variable innerhalb eines Formulars deklariert, so wird sie beim erneuten Laden des Formulars nicht neu initialisiert. Sie behält stattdessen den Wert bei, den sie beim Entladen des Formulars hatte.

Ist dies nicht gewünscht, muss sie beim Laden des Formulars mit einem Standardwert initialisiert werden. Um alle privaten Variablen eines Formulars neu zu initialisieren, kann dem Formular in der Ereignisprozedur *Unload* der Wert *Nothing* zugewiesen werden.

Nothing

```
Set FormName = Nothing
```

Die öffentliche Variable

Öffentliche Variablen werden im Allgemeinteil eines Moduls oder eines Formulars mit Hilfe des Schlüsselwortes *Public* deklariert. Sie haben einen Gültigkeitsbereich, der sich über alle Module eines Projekts erstreckt.

```
Option Explicit  
Public intAlter As Integer
```

Auf eine Variable, die mit dem Schlüsselwort *Public* im allgemeinen Teil eines Standardmoduls deklariert wurde, kann direkt, also nur durch Angabe ihres Namens zugegriffen werden. Dies gilt für alle Module des Projekts. Die entsprechende Programmanweisung hat folgendes Aussehen:

in allen Modulen sichtbar

```
intAlter = 15
```

Wurde die Variable allerdings in einem Form-Modul deklariert, so ist obige Anweisung nur innerhalb des eigenen Form-Moduls gültig. Wird aus anderen Modulen auf diese Variable zugegriffen, so muss sie wie eine Eigenschaft des Formulars behandelt werden. Die entsprechende Programmzeile sieht dann wie folgt aus:

```
FrmRente.intAlter = 15
```

3.10 Programmieren mit Variablen

Bis jetzt haben wir Variablen deklariert und mit Werten gefüllt. In einem Programm jedoch werden Variablen in Berechnungen oder für Programmentscheidungen verwendet.

Um dies zu tun, benötigen Sie Rechen- und Vergleichsbefehle. Diese werden auch als *Operatoren* bezeichnet.

3.10.1 Klassifizierung von Operatoren

Operatoren sind Rechanweisungen, die normalerweise zwei Ausdrücke miteinander verknüpfen und daraus ein Ergebnis generieren.

Operatoren können in vier verschiedene Arten untergliedert werden:

- ▶ *Arithmetische Operatoren*: Werden verwendet um Rechenoperationen durchzuführen.
- ▶ *Vergleichsoperatoren*: Benutzen Sie, wenn zwei Variablen bzw. Ausdrücke miteinander verglichen werden.
- ▶ *Logische Operatoren*: Erlauben die bitweise Verknüpfung zweier Variabler.
- ▶ *Verknüpfungsoperatoren*: Werden bei Zeichenkettenvariablen verwendet um mehrere Zeichenketten zu einer zu verbinden. (Es handelt sich dabei um den Operator &, der bereits im Abschnitt über Zeichenketten behandelt wurde.)

3.10.2 Die Rechenoperatoren

Für arithmetischen Berechnungen werden *Rechenoperatoren* verwendet. Mit ihnen werden mehrere *Operanden* entsprechend den Regeln der Algebra miteinander verknüpft. Als Operanden kommen dabei Variablen und Konstanten in Frage.

Tabelle 3.3 zeigt alle Rechenoperatoren im Überblick.

Operator	Funktion
*	Multiplikation
/	Division
+	Addition
-	Subtraktion
^	Potenzierung
\	Integerdivision
Mod	Modulo (Rest einer Division)

Tabelle 3.3:
Rechenoperatoren

Die Anwendung der Operatoren erfolgt immer nach dem gleichen Muster:

Ergebnis = Operand1 Operator Operand2 [Operator2 Operand3 ...]

Es muss aber nicht zwingend in einer Zuweisung gerechnet werden. Wenn Sie beispielsweise das Ergebnis einer Berechnung nicht speichern, sondern nur ausgeben möchten, können Sie die Berechnung auch innerhalb einer Funktion durchführen. An der Stelle, an der die Berechnung steht, wird das Ergebnis derselben eingefügt.

Das Einzige, worauf Sie bei dieser Art der Benutzung achten müssen ist, dass das Ergebnis der Berechnung auch dem Datentyp entspricht, der an der betreffenden Stelle erwartet wird. Ein Beispiel hierfür sind folgende Programmzeilen:

```
Dim intZahl1 As Integer
Dim intZahl2 As Integer
intZahl1 = 3
intZahl2 = 8
MsgBox intZahl1 * intZahl2
```

der Datentyp muss stimmen



Das Ergebnis der Berechnung ist zwar ein Datentyp *Integer*, aber die internen Konvertierungsfunktionen von Visual Basic machen aus dem *Integer* vor der Ausgabe automatisch einen *String*. Somit passt das Ergebnis und es wird der Dialog aus Abbildung 3.24 gezeigt.

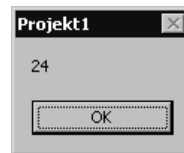


Abbildung 3.24:
Ergebnis einer Berechnung

Bis auf die Integerdivision und den Operator *Modulo* dürften die Rechenarten soweit bekannt sein. Diese beiden sind jedoch eine Spezialisierung der Division, die für die Programmierung sehr sinnvoll ist.

ganzzahlige Division

Die *Integerdivision* (Operator `\`) liefert als Ergebnis einen ganzzahligen Wert, d.h. die Nachkommastellen, oder anders ausgedrückt: Der Rest der Division wird abgeschnitten.

Rest der ganzzahligen Division

Der Operator *Modulo* ist eine Ergänzung zur Integerdivision, da er den Rest einer Division ermittelt und als Ergebnis zurückliefert. Ein Beispiel zeigt die Funktionsweise:



```
Dim intZahl1 As Integer
Dim intZahl2 As Integer
intZahl1 = 8
intZahl2 = 3
MsgBox "IntegerDivision " & vbTab & intZahl1 \ intZahl2 & vbCrLf &
"Rest" & vbTab & vbTab & intZahl1 Mod intZahl2
```

Wie Sie in Abbildung 3.25 sehen, ist das Ergebnis der Integerdivision zwei. Das Ergebnis der Modulo-Operation, also der ganzzahlige Rest, der bei einer Integerdivision verbleibt, ist ebenfalls zwei.

Abbildung 3.25:
Integerdivision und
Modulo



Bei der Ausgabe des Ergebnisses wurde ein kleiner Trick verwendet, um die Ergebnisse untereinander darzustellen. Es wurde die vordefinierte Konstante `vbTab` verwendet, um die Ergebnisse auf der gleichen Spalte darzustellen. Mit den vordefinierten Variablen `vbCrLf` und `vbTab` sind Sie in der Lage, die Ausgaben einer *MsgBox*-Anweisung zu formatieren.

Priorität von Operatoren

Punkt- vor Strichrechnung

Sie kennen sicher die Regel *Punkt- vor Strichrechnung*. Diese ist notwendig, um sicherzustellen, dass bei Berechnungen, in denen hintereinander mehrfach Operanden und Operatoren verwendet werden, ein eindeutiges Ergebnis zu erwarten ist.

Klammern

Sie können diesem Problem in Visual Basic auch durch das Setzen von Klammern entgegen. Aber wie in der normalen Algebra, gibt es in Visual Basic Prioritäten für jeden Operator.

Die Hierarchie entspricht in Visual Basic weitgehend der oben genannten *Punkt-vor-Strich-Regel*. Alle Operatoren und deren Priorität finden Sie in Tabelle 3.4.

Priorität	Operator	Funktionsbeschreibung
1		Funktionsaufrufe, Klammern
2	^	Potenzierung
3	-	Vorzeichenoperator
4	* /	Multiplikation, Division
5	\ Mod	Integerdivision, Modulo
6	+ -	Addition, Subtraktion
7	=, <>, <, >, <=, >=	Vergleichsoperatoren
8	Not	Invertierung aller Bits eines Operanden
9	And	Logische UND-Verknüpfung
10	Or	Logische ODER-Verknüpfung
11	Xor	Logische XOR-Verknüpfung
12	Eqv	Äquivalenz-Verknüpfung
13	Imp	Implikations-Verknüpfung

Tabelle 3.4:
Operatorpriorität

Es gelten folgende Regeln: Je kleiner die Zahl in Tabelle 3.4, desto höher die Priorität des zugehörigen Operators. Falls in einer Berechnung mehrere Operatoren mit gleicher Priorität hintereinander stehen, so wird von links nach rechts gerechnet.

Falls Sie in einer Berechnung andere als die durch Visual Basic vorgegebenen Prioritäten benötigen, müssen Sie Klammern verwenden.

Folgende Beispiele sollen dieses Verhalten nochmals verdeutlichen:

$10 / 5 + 2 * 3$

Das Ergebnis dieser Berechnung ist *acht*. Nach den Prioritätsregeln löst Visual Basic den Term auf, in dem zunächst $10 / 5$ und $2 * 3$ gerechnet wird. Anschließend werden die Ergebnisse nach der ursprünglichen Formel weiter gerechnet, also $2+6 = 8$.

Ein weiteres Beispiel zeigt die Regel von *links nach rechts*:

$10 / 5 * 3$

Das Ergebnis dieser Berechnung ist sechs. Die Operatoren $/$ und $*$ haben laut Tabelle 3.4 gleiche Priorität. Daher wird zunächst (von links nach rechts) $10 / 5$ gerechnet. Anschließend wird das Ergebnis dieser Berechnung, nämlich 2 mit 3 multipliziert.

3.10.3 Übung: Berechnung eines Rabatts

Machen Sie ein Programm, welches auf einen Preis einen Rabatt berechnet und die nach Abzug des Rabatts zu zahlende Summe ausgibt. Der Anwender soll die Möglichkeit haben einen Einzelpreis, eine Menge und einen variablen Rabatt in Prozent einzugeben.

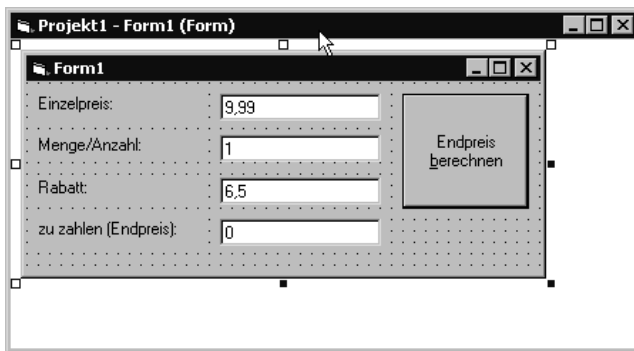


Lösung

Für die Lösung dieser Übung wird zunächst die Oberfläche gestaltet. In Visual Basic wird hierzu ein neues Standardprojekt geöffnet. Wählen Sie dazu im Dialog NEUES PROJEKT die Auswahl STANDARD-EXE.

Fügen Sie der Oberfläche für die notwendigen Eingaben und Ausgaben vier *TextBox* hinzu. Zudem benötigen wir für die Platzierung eines erläuternden Textes vier *Label* und für das Anstoßen der Berechnung einen *CommandButton*. Die Oberfläche meiner Lösung sieht dann aus wie in Abbildung 3.26.

Abbildung 3.26:
Berechnung eines
Endpreises mit
Rabatt



Wie Sie in Abbildung 3.26 sehen können, wurden die Eingabeelemente bereits mit Werten gefüllt. Dies hat zum einen den positiven Effekt, dass Sie bereits einen Defaultwert vorgeben ohne eine Zeile zu programmieren. Gerade bei einem Wert wie Rabatt muss später in vielen Fällen keine Eingabe erfolgen, weil der Defaultwert bereits korrekt ist. Zum anderen hat der Anwender durch den Defaultwert einen Hinweis, welche Art von Eingabe von ihm verlangt wird.

Jetzt wird die eigentliche Lösung programmiert: die Berechnung des Endpreises. Hierzu wird der Code-Editor für die Ereignisroutine *Click* des *CommandButtons* geöffnet.

Konvertierungsfunktion aufrufen

Zunächst werden die notwendigen Variablen deklariert und mit den Werten aus den Eingabe-Textboxen initialisiert. Da das Steuerelement *TextBox* einen String zurückliefert, wird jeweils die Konvertierungsfunktion *Cdbl* aufgerufen. Diese Funktion wandelt einen Wert in eine Zahl des Datentyps *Double* um.

Der entsprechende Programmcode sieht wie folgt aus:

```
Private Sub BT_rechnen_Click()  
Dim dblEinzelpreis As Double  
Dim dblMenge As Double  
Dim dblRabatt As Double  
dblEinzelpreis = Cdbl(TX_Einzelpreis.Text)  
dblMenge = Cdbl(TX_Menge.Text)  
dblRabatt = Cdbl(TX_Rabatt.Text)
```

Nachdem alle für die Berechnung notwendigen Variablen initialisiert sind, wird die eigentliche Berechnung durchgeführt:

```
TX_Endpreis.Text = dblEinzelpreis * intMenge - dblEinzelpreis *
dblMenge / 100 * dblRabatt
End Sub
```

Wie Sie sehen, wird für den Endpreis keine eigene Variable deklariert. Stattdessen wird der Endpreis direkt in der TextBox *TX_Endpreis* ausgegeben. Hierbei findet eine automatische Konvertierung des Ergebnisses durch Visual Basic statt.

In der Berechnung selbst werden keine Klammern benötigt, da die normale Priorität der Operatoren (Punkt vor Strich) ausreicht. So wird von Visual Basic zuerst der Term vor dem Minuszeichen, dann der Term nach dem Minuszeichen und am Ende die Subtraktion ausgeführt. In den Einzeltermen wiederum wird jeweils von links nach rechts gerechnet, da die Operatoren *** und */* gleiche Priorität haben.

Wenn Sie das Programm starten, erscheint zunächst die Eingabemaske mit den Defaultwerten, die im Formulareditor beim Design der Oberfläche eingegeben wurden. In Abbildung 3.27 wurden Eingaben gemacht und eine Berechnung durchgeführt.

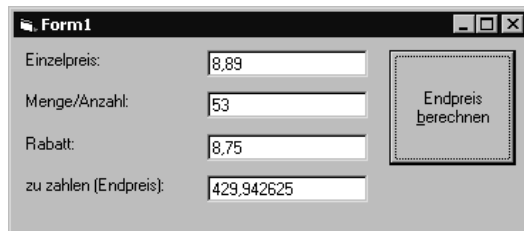


Abbildung 3.27:
Test des
Programms

Da es bei diesem Programm lediglich um die Anwendung der Rechenoperatoren ging, wurde bei der Lösung der Übung auf jedwede Fehlerbehandlung verzichtet. Wenn Sie professionelle Programme schreiben, die auch verkauft werden sollen, müssen potenzielle Fehleingaben berücksichtigt und abgefangen werden.



Versuchen Sie doch einmal obige Berechnung durchzuführen und anstatt des Kommas einen Punkt einzugeben. Oder was passiert, wenn Sie mit sehr großen Preisen und Mengen rechnen?

3.10.4 Die Vergleichsoperatoren

Das Ergebnis einer Rechenoperation kann im Prinzip in einem beliebig großen Wertebereich liegen. Bei Vergleichsoperatoren ist zwar die Syntax im Prinzip gleich, das Ergebnis hat aber in jedem Fall nur zwei mögliche Werte, nämlich *True* oder *False*, also *Wahr* oder *Unwahr*.

**für Programm-
entscheidungen**

Vergleichsoperatoren werden daher auch äußerst selten verwendet, um das Ergebnis einer Variablen zuzuweisen. Stattdessen wird das Ergebnis einer Vergleichsoperation normalerweise für eine Programmentscheidung verwendet

In Tabelle 3.5 werden alle Vergleichsoperatoren von Visual Basic gelistet.

*Tabelle 3.5:
Vergleichs-
operatoren*

Operator	Funktion
=	gleich
<	kleiner
<=	kleiner gleich
>	größer
>=	größer gleich
<>	ungleich

**Ergebnis ist True
oder False**

Vergleichsoperatoren werden sowohl mit numerischen als auch mit alphanumerischen Operanden verwendet. Bei einem Vergleich mit numerischen Operanden ist die Ordnung der Zahlen bekannt. Die Zahl -10 ist beispielsweise kleiner als 0. Folgender Vergleich würde also als Ergebnis den Wert *True* annehmen:

`-10 < 0 (= True)`

Bei alphanumerischen Variablen ist die Sache schon nicht mehr so eindeutig. Was wäre das Ergebnis bei folgendem Vergleich?

`»-10« < »0« (= ?)`

Die Visual Basic Online-Hilfe sagt zu diesem Thema, dass eine Zeichenkette dann als kleiner gehandelt wird, wenn das erste unterschiedliche Zeichen der Zeichenkette im Alphabet vor dem Zeichen des Vergleichstrings liegt. Nun ist das bei der Zeichenkette »-10« etwas schwierig, denn wer kann schon sagen, wo sich im Alphabet das Zeichen »-« befindet?

Tatsächlich wird der Vergleich anhand einer Standard-ASCII-Tabelle gemacht. Dort sind alle Zeichen, die auf dem PC standardmäßig verwendet werden, eingetragen, wobei jedem Zeichen eine Zahl im Bereich bis 255 zugewiesen ist.

Die Reihenfolge der Zeichen in der ASCII-Tabelle, können sich bei den Sonder- oder länderspezifischen Zeichen unterscheiden. Aber im Prinzip gilt Folgendes:

- ▶ Die Zahlen befinden sich in der ASCII-Tabelle vor den Großbuchstaben.
- ▶ Die Großbuchstaben befinden sich in der ASCII-Tabelle vor den Kleinbuchstaben.
- ▶ Die Reihenfolge der Zahlen und Buchstaben ist von 0 bis 9 und von A-Z bzw. a-z.
- ▶ Bei allen anderen Zeichen ist ein Nachschauen in einer ASCII-Tabelle empfehlenswert.

Der Vergleich im obigen Beispiel ist *wahr*, denn das *Minus*-Zeichen befindet sich in der ASCII-Tabelle vor dem Zeichen 0.

3.10.5 Übung: Verwendung von Vergleichsoperatoren



Ermitteln Sie von zwei eingegebenen alphanummerischen Werten den größeren.

Verwenden Sie zur Lösung dieser Übung die *if*-Anweisung in folgender Form:

If Vergleich *then* Anweisung

Lösung

Um diese Übung zu lösen, wird ein Formular benötigt, welches die Eingabe der zwei alphanummerischen Werte erlaubt. Zudem sollte sich auf dem Formular ein *CommandButton* befinden, über welchen der Vergleich angestoßen wird.

In Abbildung 3.28 sehen Sie ein solches Formular:

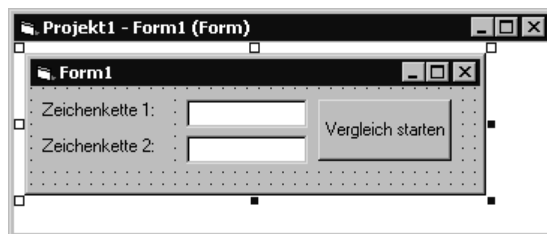


Abbildung 3.28:
Formular des
Programms
Vergleichstest

Mit folgenden Programmzeilen in der Ereignisroutine *Click* des *CommandButtons* *Command1* wird festgestellt, welcher String größer ist:

```
Private Sub Command1_Click()  
If TX_String1.Text > TX_String2.Text Then MsgBox TX_String1.Text & "  
ist größer als " & TX_String2.Text  
If TX_String2.Text > TX_String1.Text Then MsgBox TX_String2.Text & "  
ist größer als " & TX_String1.Text  
End Sub
```

In beiden *If*-Anweisungen werden die eingegebenen Zeichenketten mit dem Vergleichsoperator *>* (größer) überprüft. Falls das Ergebnis des Vergleichs *wahr* ist, wird ein Dialog gezeigt.

Was passiert allerdings, wenn die beiden Zeichenketten gleich sind?

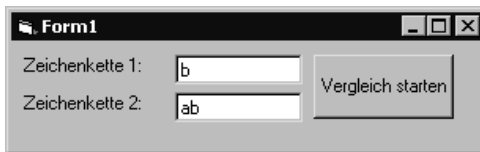
In diesem Fall passiert nichts. Die Ereignisroutine wird in beiden Fällen beim Vergleich das Ergebnis *falsch* ermitteln und folgerichtig keine Ausgaben machen.

Um diesen Zustand zu verhindern, wird ein weiterer Vergleich eingeführt, der den dritten Zustand (gleiche Zeichenketten) ermittelt und ebenfalls eine Ausgabe macht.

```
Private Sub Command1_Click()  
If TX_String1.Text > TX_String2.Text Then MsgBox TX_String1.Text & "  
ist größer als " & TX_String2.Text  
If TX_String2.Text > TX_String1.Text Then MsgBox TX_String2.Text & "  
ist größer als " & TX_String1.Text  
If TX_String1.Text = TX_String2.Text Then MsgBox TX_String1.Text & "  
und " & TX_String1.Text & " sind gleich."  
End Sub
```

Sie können jetzt das Programm starten und beliebige Eingaben miteinander vergleichen. In Abbildung 3.29 sehen Sie das Programm mit den Eingaben b als Zeichenkette 1 und ab als Zeichenkette 2.

Abbildung 3.29:
Das Programm in
Aktion



Das Ergebnis des Vergleichs von b und ab sehen Sie in Abbildung 3.30.

Abbildung 3.30:
Das Ergebnis des
Vergleichs



Obiges Beispiel zeigt nochmals die Art des Vergleichs mit den Vergleichsoperatoren bei Zeichenketten. Die Länge der Zeichenkette ist nicht von Bedeutung, entscheidend für das Ergebnis ist lediglich das erste unterschiedliche Zeichen.

3.10.6 Die logischen Operatoren

Mit logischen Operatoren werden die Operanden bitweise verknüpft. Am häufigsten werden diese Operatoren jedoch angewendet, um in Entscheidungsstrukturen zwei Ausdrücke logisch zu verknüpfen.

Tabelle 3.6 zeigt eine Übersicht der wichtigsten logischen Operatoren.

Operator	Funktion
And	Logische UND-Verknüpfung
Or	Logische ODER-Verknüpfung
Xor	Logische exklusive ODER-Verknüpfung
Not	Logisches NICHT
Eqv	Äquivalenz

Tabelle 3.6:
Logische
Operatoren

Um das Ergebnis einer logischen Operation zu ermitteln, wird eine *Wahrheitstabelle* benötigt. In der Wahrheitstabelle werden in der ersten Spalte der Wert von Operand1, in der zweiten Spalte der Wert von Operand2 und in der dritten Spalte das jeweilige Ergebnis dargestellt.

**Ergebnis aus
Wahrheitstabelle**

Werden die logischen Operatoren zur Verknüpfung zweier Ausdrücke verwendet, so kann das Ergebnis direkt aus der Wahrheitstabelle abgelesen werden. Werden jedoch zwei numerische Werte bitweise miteinander verknüpft, so kann in der Wahrheitstabelle der Wert True für den Bitwert 1, der Wert False für den Bitwert 0 gesetzt werden. Im Ergebnis einer bitweisen Verknüpfung wird das Bit an der Stelle dann gemäß der Wahrheitstabelle der Operators gesetzt.

bitweise Verknüpfung

Folgendes Beispiel zeigt eine bitweise Verknüpfung zweier Zahlen. Diese werden zur Demonstration als Binärzahlen dargestellt.

```
Operand1      1010
Operand2      1000
Ergebnis AND 1000
```



Die Bits des Ergebnisses wurden aus der Wahrheitstabelle des Operators AND (Tabelle 3.7) ermittelt.

Die Wahrheitstabelle des Operators And

Operand 1	Operand 2	Ergebnis
False	False	False
False	True	False
True	False	False
True	True	True

Tabelle 3.7:
Wahrheitstabelle
des Operators And

Aus dieser Wahrheitstabelle folgt, dass ein mit *And* zusammengesetzter Ausdruck nur dann *wahr* ist, wenn beide Einzelbedingungen *wahr* sind.

Die Wahrheitstabelle des Operators Or

Tabelle 3.8:
Wahrheitstabelle
des Operators Or

Operand 1	Operand 2	Ergebnis
False	False	False
False	True	True
True	False	True
True	True	True

Die logische Verknüpfung mit *Or* ergibt immer dann als Ergebnis *True*, wenn einer der beiden Operanden *True* ist.

Die Wahrheitstabelle des Operators XOr

Tabelle 3.9:
Wahrheitstabelle
des Operators XOr

Operand 1	Operand 2	Ergebnis
False	False	False
False	True	True
True	False	True
True	True	False

Der Operator *XOr* ergibt nur dann *True*, wenn einer der beiden Operanden *True* und der andere *False* ist.

Die Wahrheitstabelle des Operators Not

Der Operator *Not* hat eine Sonderstellung, da er nicht zwei Operanden miteinander verknüpft, sondern einen Operanden negiert. Seine Wahrheitstabelle ist daher sehr kurz.

Tabelle 3.10:
Wahrheitstabelle
des Operators Not

Operand 1	Ergebnis
False	True
True	False

Die Wahrheitstabelle des Operators Eqv

Tabelle 3.11:
Wahrheitstabelle
des Operators Eqv

Operand 1	Operand 2	Ergebnis
False	False	True
False	True	False
True	False	False
True	True	True

Der Operator *Eqv* ist die Umkehrung des Operators *Xor*. Er liefert *True*, wenn beide Operanden *True* oder *False* sind.

3.10.7 Übung: Verwendung von logischen Operatoren



Prüfen Sie, ob eine Zahl im Bereich zwischen eins und zwanzig liegt. Verwenden Sie hierzu folgende if-Struktur:

```
If vergleich then
    Anweisungen
Else
    Anweisungen
End if
```

Bei dieser Struktur werden, wenn der Vergleich *wahr* ist, die Anweisungen hinter *then* ausgeführt. Ist der Vergleich *unwahr*, so werden die Anweisungen hinter *else* ausgeführt.

Lösung

Zunächst wird wie gehabt eine Programmoberfläche designed. Da die Grenzen der Prüfung nicht variabel sind, benötigen wir lediglich ein Eingabefeld. Das Formular wird also recht überschaubar und sollte in etwa so aussehen wie in Abbildung 3.31.

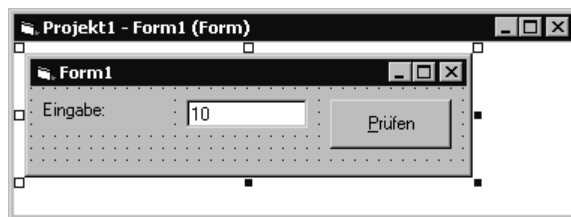


Abbildung 3.31:
Die Oberfläche des
Programms

Damit bei Betätigen der Schaltfläche auch etwas passiert, muss jetzt die Ereignisroutine *Click* des *Commandbuttons* mit Leben erfüllt werden. Folgende Programmzeilen erfüllen die Aufgabe:

```
Private Sub Command1_Click()
    Dim lngEingabe As Long
    lngEingabe = CLng(TX_Eingabe.Text)
    If lngEingabe > 0 And lngEingabe < 21 Then
        MsgBox lngEingabe & " ist zwischen 1 und 20."
    Else
        MsgBox lngEingabe & " ist nicht zwischen 1 und 20."
    End If
End Sub
```

Zunächst wird eine Variable deklariert und mit dem eingegebenen Wert gefüllt. Um den eingegebenen Text in eine Zahl zu wandeln, wird die Funktion *CLng* aufgerufen. Diese wandelt einen Wert in eine Zahl des Datentyps *Long* um.

3 Workshop: Variablen und Konstanten

Der Vergleich in der if-Anweisung enthält die eigentliche Lösung. Da es keinen Vergleichsoperator gibt, der »zwischen«-ermittelt, muss hier mit zwei verknüpften Termen gearbeitet werden. Dazu wird die Aufgabenstellung »muss zwischen 1 und 20« liegen umformuliert in »muss größer sein als 0 und kleiner als 21«.

Diese Formulierung kann jetzt direkt in Programmcode umgesetzt werden:

```
IngEingabe > 0 And IngEingabe < 21
```



Beachten Sie bitte bei diesem Term die Priorität der Operatoren. Der logische Verknüpfungsoperator hat eine kleinere Priorität als die beiden Vergleichsoperatoren. Daher löst Visual Basic den Term auf, indem zuerst die Vergleiche gemacht werden. Anschließend werden die Ergebnisse der beiden Vergleiche gemäß der Wahrheitstabelle des logischen Operators AND miteinander verknüpft.

Abbildung 3.32:
Eingabeformular
mit Defaultwert

Abbildung 3.32 zeigt die Form nach dem Programmstart. Die Auflösung des Terms für den Defaultwert 10 ergibt sich nach folgendem Muster:

1. Visual Basic ermittelt das Ergebnis von $10 > 0 = \text{wahr}$
2. Der Term $\text{IngEingabe} < 21$ wird aufgelöst. $10 < 21$ ist ebenfalls *wahr*.
3. Das Ergebnis der beiden Terme wird mit dem AND-Operator verknüpft. *wahr AND wahr* ergibt WAHR.

Somit hat Visual Basic festgestellt, dass die Bedingung insgesamt *wahr* ist, der Wert 10 also zwischen 1 und 20 liegt. Als Ergebnis wird daher der Dialog aus Abbildung 3.33 ausgegeben.

Abbildung 3.33:
Bedingung ist wahr

Wird die gleiche Berechnung für den Wert -3 durchgeführt, so ermittelt der Vergleich den Wert *falsch*. Nach obigem Muster wird dies folgendermaßen ermittelt:

1. Visual Basic ermittelt das Ergebnis von $-3 > 0 = \text{unwahr}$
2. Der Term $\text{IngEingabe} < 21$ wird aufgelöst. $-3 < 21$ ist *wahr*.

3. Das Ergebnis der beiden Terme wird mit dem AND-Operator verknüpft. *Unwahr AND wahr* ergibt unwahr.

Somit wurde festgestellt, dass -3 nicht im Bereich zwischen 1 und 20 liegt. Es wird die Anweisung des Else-Zweiges ausgeführt. Ausgegeben wird der Dialog aus Abbildung 3.34.



Abbildung 3.34:
Bedingung ist nicht
wahr, Ausgabe im
Else-Zweig

4

Workshop: Steuerung des Programmablaufs

In einem Programm ist es zumeist notwendig, abhängig von Benutzereingaben, zur Laufzeit ermittelten Ergebnissen oder sonstigen Ereignissen unterschiedliche Aufgaben durchzuführen. Hierzu sind Entscheidungsstrukturen notwendigen, die den Programmablauf steuern.

In den bisherigen Übungen wurde bereits eine Entscheidungsstruktur kurz vorgestellt. Es handelte sich um eine *If*-Anweisung, die dazu verwendet wurde, abhängig von der gestellten Aufgabe eine Benutzereingabe zu prüfen und, je nach Ergebnis der Prüfung, eine Ausgabe zu machen.

Ein weiterer Grund, den Programmablauf zu steuern, besteht dann, wenn in Ihrem Programm wiederholt die gleichen Anweisungen ausgeführt werden müssen. Sie müssen in diesem Fall nicht die gleiche Programmanweisung sehr oft untereinander schreiben, sondern verwenden stattdessen einfach eine *Programmschleife*.

In den folgenden Abschnitten werden die Programmanweisungen vorgestellt, die für die Steuerung des Programmablaufs in Visual Basic zur Verfügung stehen.

Entscheidungen

Wiederholungen

4.1 Entscheidungsstrukturen

Immer dann, wenn der weitere Ablauf Ihres Programms abhängig ist von einem aktuellen Zustand, benötigen Sie eine Entscheidungsstruktur. Visual Basic bietet einige Varianten, die zweckentsprechend eingesetzt werden können.

4.1.1 Die If...Then-Anweisung

Die *If...Then*-Anweisung ist eine einfache Entscheidungsstruktur, die eine *Wenn...Dann*-Entscheidung trifft. In den Übungen zu Vergleichsoperatoren wurde bereits die einfachste Form der *If...Then*-Anweisung verwendet. Es gibt zwei mögliche Syntaxformen der *If...Then*-Anweisung:

```
If Bedingung Then Anweisungszeile [Else elseAnweisungszeile]
```

```
If Bedingung Then  
[Anweisungen]  
[ElseIf Bedingung-n Then  
[elseifAnweisungen] ...  
[Else  
[elseAnweisungen]]  
End If
```

Wenn...Dann-Entscheidung

Syntax 1

Syntax 2

In *Syntax 1* wird, wenn die Bedingung *wahr* ist, die Anweisungszeile hinter *Then* ausgeführt. Im Unterschied zu *Syntax 2* kann an dieser Stelle kein mehrzeiliger Anweisungsblock stehen. Müssen mehrere Anweisungen ausgeführt werden, können diese durch Doppelpunkte getrennt hintereinander geschrieben werden. Es empfiehlt sich allerdings in einem solchen Fall, doch besser die einfachste Form der *Syntax 2* zu verwenden.

Ist die Bedingung *unwahr*, so wird die Anweisungszeile hinter dem Schlüsselwort *Else* ausgeführt. Fehlt die *Else*-Anweisung, so setzt das Programm seine Ausführung hinter der *End If*-Anweisung fort.

Mit der *If...Then*-Anweisung in der Form *Syntax 2* kann ein zusätzliches Schlüsselwort verwendet werden. Es handelt sich hierbei um das Schlüsselwort *Elseif*.

Elseif wird verwendet, um innerhalb einer *If...Then*-Struktur mehrere Bedingungen abzufragen. Das *Elseif* tritt hierbei an die Stelle des *Else* und kann im Gegensatz zu diesem beliebig oft wiederholt werden. Eine entsprechende Anweisung würde wie folgt aussehen:



```
If strEingabe = "e" Then
    MsgBox "Auswahl e wurde eingegeben"
Elseif strEingabe = "a" Then
    MsgBox "Auswahl a wurde eingegeben"
Elseif strEingabe = "m" Then
    MsgBox "Auswahl m wurde eingegeben"
.
.
.
Else
    MsgBox "Die Auswahl ist ungültig"
End If
```

Visual Basic wertet die Ausdrücke der einzelnen *If*- und *Elseif*-Bedingungen von oben nach unten aus. Sobald eine Bedingung als *Wahr* erkannt wird, wird der zugehörige Anweisungsblock ausgeführt. Danach wird die Verarbeitung hinter der *End-If*-Anweisung fortgesetzt. Es wird also immer nur ein Anweisungsblock ausgeführt.



4.1.2 Übung: Verwendung der *If...Then*-Struktur

1. Schreiben Sie ein Programm, welches ermittelt, ob ein eingegebener Buchstabe ein Kleinbuchstabe ist.
2. Erweitern Sie das Programm um eine Schaltfläche, die ermittelt, ob der eingegebene Buchstabe ein Kleinbuchstabe, ein Großbuchstabe oder eine Zahl ist.

3. Erweitern Sie das Programm um eine Schaltfläche, die zusätzlich zu obigen Prüfungen eine Ausgabe macht, wenn keiner der untersuchten Bereiche zutrifft.

Lösung

Bei den Teilübungen 2 und 3 handelt es sich um eine Erweiterung des Programms für Teilübung 1. Jede Teilübung wird mit einer eigenen Schaltfläche gestartet. Wir benötigen also eine Programmoberfläche, die die Eingabe eines Zeichens erlaubt und drei verschiedenartige Prüfungen auf dieses Zeichen machen kann.

Mein Vorschlag für diese Oberfläche besteht aus einer Textbox für die Eingabe des Zeichens und drei Schaltflächen, wie in Abbildung 4.1 zu sehen.

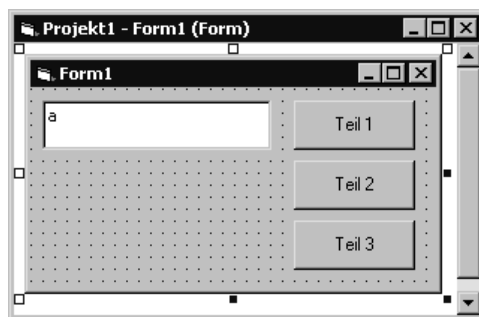


Abbildung 4.1:
Oberfläche des
Programms

Um Teilübung 1 zu programmieren, wird per Doppelklick auf die Schaltfläche mit der Beschriftung Teil 1 der Programmierer geöffnet. Visual Basic legt automatisch eine leere Ereignisroutine *Click* des zugehörigen Commandbuttons *BT_Aufgabe1* an.

In diese wird folgende Programmzeile eingegeben:

```
Private Sub BT_Aufgabe1_Click()  
    If TX_Eingabe.Text >= "a" And TX_Eingabe.Text <= "z" Then MsgBox  
    TX_Eingabe.Text & " ist ein Kleinbuchstabe."  
End Sub
```

Es handelt sich dabei um die einfachste, einzeilige Form der *If...Then*-Anweisung. Hinter dem Schlüsselwort *If* steht das Herzstück dieser Anweisung, die Bedingung. Wenn sie *wahr* ist, wird ein Text ausgegeben, der besagt, dass der eingegebene Buchstabe ein Kleinbuchstabe ist.

Kleinbuchstaben sind ein bestimmter Bereich in der ASCII-Tabelle. Die Reihenfolge der kleinen Buchstaben hält sich an das Alphabet. Daher kann festgestellt werden, dass sich alle kleinen Buchstaben zwischen den beiden Buchstaben a und z befinden (Umlaute werden in diesem Fall außer Acht gelassen).

Die Abfrage muss also lediglich feststellen, ob der eingegebene Buchstabe größer oder gleich a und kleiner oder gleich z ist. Die Bedingung ist demnach in Visual Basic-Programmcode ausformuliert:

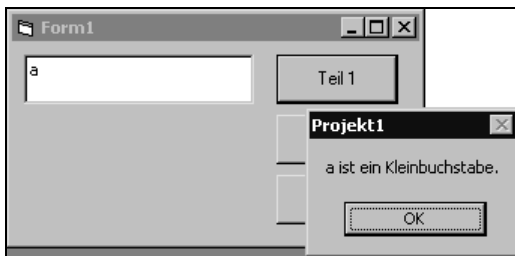
```
TX_Eingabe.Text >= "a" And TX_Eingabe.Text <= "z"
```

Priorität der Operatoren

Da die Eigenschaft *Text* einer *TextBox* einen String zurückliefert, wird keine Zwischenvariable benötigt, die die Eingabe in einen für den Vergleich brauchbaren Datentyp umwandelt. Klammern werden ebenfalls nicht benötigt, da die Vergleichsoperatoren `>=` und `<=` eine höhere Priorität haben als der logische Operator *AND*.

Wenn Sie das Programm starten und einen kleinen Buchstaben eingeben, erhalten Sie auch folgerichtig einen Dialog als Antwort, der Ihnen mitteilt, dass Sie einen Kleinbuchstaben eingegeben haben. Eine solche Ausgabe sehen Sie in Abbildung 4.2.

Abbildung 4.2:
Ausgabe von
Teilübung 1



Was aber passiert, wenn Sie in die *TextBox* den Buchstaben A eingeben? Nun, es passiert überhaupt nichts. Ein normaler Anwender würde bei dieser Reaktion sicher annehmen, dass das Programm nicht funktioniert oder gar abgestürzt ist. Dieser Mangel wird umgehend durch die Programmierung der Teilübung 2 in der Ereignisroutine des Commandbuttons *BT_Aufgabe2* beseitigt.

```
Private Sub BT_Aufgabe2_Click()  
    If TX_Eingabe.Text >= "a" And TX_Eingabe.Text <= "z" Then  
        MsgBox TX_Eingabe.Text & " ist ein Kleinbuchstabe."  
    ElseIf TX_Eingabe.Text >= "A" And TX_Eingabe.Text <= "Z" Then  
        MsgBox TX_Eingabe.Text & " ist ein Großbuchstabe."  
    ElseIf TX_Eingabe.Text >= "0" And TX_Eingabe.Text <= "9" Then  
        MsgBox TX_Eingabe.Text & " ist eine Zahl."  
    End If  
End Sub
```

In diesem Fall sind insgesamt drei Bedingungen zu prüfen. Die Prüfung auf Großbuchstaben und Zahlen erfolgt analog, wie es in Teilübung 1 bereits für Kleinbuchstaben durchgeführt wurde.



Die Übung könnte auch mit drei unabhängigen *If...Then*-Anweisungen durchgeführt werden. Gewählt wurde aber eine Lösung mit einer *If...Then...Elseif*-

Struktur. Bei drei unabhängigen *If...Then*-Konstrukten würde Visual Basic auch bei Eingabe eines Kleinbuchstabens die Prüfung auf Großbuchstaben und Zahlen durchführen.

Da ein Kleinbuchstabe jedoch kein Großbuchstabe oder ein Zahl sein kann, ist diese Prüfung überflüssig. Sobald eine der Bedingungen sich als *wahr* herausstellt, ist es nicht mehr notwendig, weitere Prüfungen durchzuführen, da ein eindeutiges Ergebnis vorliegt.

Die *If...Then...Else*-Struktur trägt diesem Rechnung. Sobald eine der Bedingungen sich als *wahr* herausstellt, wird der zugehörige Anweisungsblock ausgeführt und danach die weitere Programmausführung hinter der *End If*-Anweisung fortgesetzt. Auf diese Weise werden keine unnötigen Prüfungen durchgeführt.

Wenn in einem Ihrer Programme eine *If...Then...ElseIf*-Anweisung vorkommt und die Anwendung zeitkritisch ist, sollten Sie die Reihenfolge Ihrer Prüfungen nach der Häufigkeit ordnen. Damit können Sie Ihr Programm beschleunigen, da Sie die Wahrscheinlichkeit erhöhen, dass Visual Basic für die Anweisung nur eine bzw. nur wenige Prüfungen machen muss.

keine überflüssigen Prüfungen

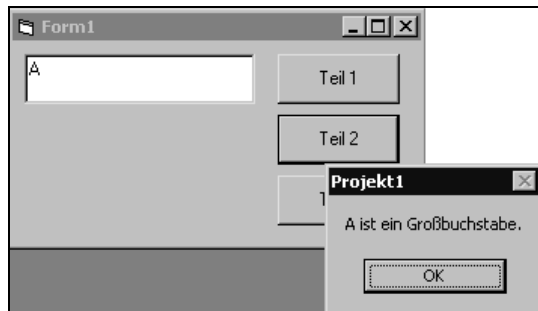


Abbildung 4.3:
Teilübung 2 kann auch Großbuchstaben erkennen

In Abbildung 4.3 und Abbildung 4.4 sehen Sie, dass sowohl Großbuchstaben als auch Zahlen korrekt erkannt werden.

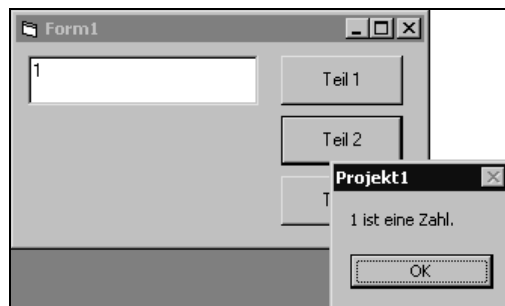


Abbildung 4.4:
... und Zahlen

Bekommt der Anwender jetzt bei jeder möglichen Eingabe eine Antwort von unserem Programm?

Die Antwort ist: Nein. Tatsächlich gibt es in der ASCII-Tabelle noch jede Menge Zeichen, die mit diesen Prüfungen nicht abgefangen werden. Es handelt sich hierbei um Sonderzeichen, Umlaute, teilweise auch Steuerzeichen wie Tabulator oder Linefeed. Zudem könnte ein Anwender auf die Idee kommen keine Eingabe zu machen.

Daher ist die Erweiterung des Programms, wie in Teilübung 3 verlangt, auf jeden Fall sinnvoll. Es wird hierzu folgender Programmcode in der Ereignisroutine der dritten Schaltfläche benötigt.

```
Private Sub BT_Aufgabe3_Click()  
    If TX_Eingabe.Text >= "a" And TX_Eingabe.Text <= "z" Then  
        MsgBox TX_Eingabe.Text & " ist ein Kleinbuchstabe."  
    ElseIf TX_Eingabe.Text >= "A" And TX_Eingabe.Text <= "Z" Then  
        MsgBox TX_Eingabe.Text & " ist ein Großbuchstabe."  
    ElseIf TX_Eingabe.Text >= "0" And TX_Eingabe.Text <= "9" Then  
        MsgBox TX_Eingabe.Text & " ist eine Zahl."  
    Else  
        MsgBox TX_Eingabe.Text & " ist weder ein Kleinbuchstabe noch ein  
        Großbuchstabe noch eine Zahl."  
    End If  
End Sub
```

Sie werden feststellen, dass lediglich der *Else*-Zweig hinzugekommen ist. Dieser erlaubt, ohne weitere Bedingung auf alle verbleibenden Eingaben zu reagieren.



Es lohnt sich in den meisten Fällen, eine *If...Then...ElseIf*-Anweisung mit einem *Else*-Zweig zu versehen. Auch wenn Sie glauben, dass dieser Zweig nie aktiviert wird, sollten Sie zumindest eine informative Ausgabe machen, damit Sie später feststellen können, warum Ihr Programm doch in diesem Zweig gelandet ist. Bedenken Sie dabei, dass Ihr Programm durch den *Else*-Zweig nicht langsamer wird, denn wenn diese Bedingung nie eintritt, wird sie auch nie ausgeführt.

Mit dem *Else*-Zweig kann der Anwender jetzt tatsächlich beliebige Eingaben machen, er wird in jedem Fall eine Antwort bzw. Reaktion des Programms sehen.

In Abbildung 4.5 sehen Sie die Ausgabe des *Else*-Zweiges für den Fall, dass keine Eingabe erfolgt ist bzw. der Defaultwert aus der TextBox gelöscht wurde.



4.1.3 Übung: Verhindern der Division durch 0 mit *If...Then*

Schreiben Sie ein Programm, welches die Division zweier Zahlen erlaubt. Verhindern Sie durch eine *If...Then*-Struktur die Division durch 0.

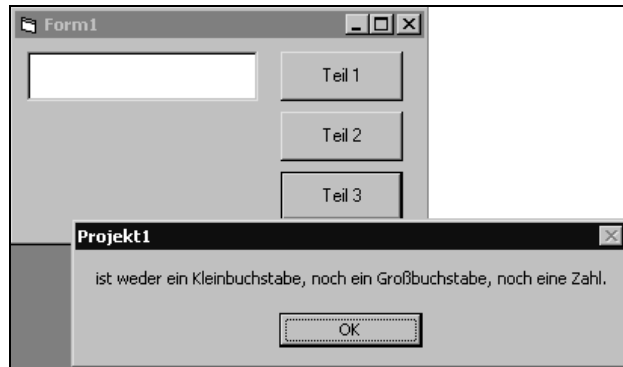


Abbildung 4.5:
Der Else-Zweig
macht den Rest

Lösung

Ein Beispiel zeigt den sinnvollen Einsatz eines Vergleichs. Der Vergleich soll in einem Beispielprogramm verhindern, dass eine Division durch Null stattfindet.

Sie sollten hierzu in der Entwicklungsumgebung ein Projekt mit einer Form anlegen. Auf dieser Form werden zwei *Textboxen* platziert, die der Eingabe der beiden Operanden dienen.

Zudem wird ein *Label* benötigt, welches die Ausgabe des Ergebnisses aufnimmt. Eine Schaltfläche wird verwendet, um nach der Eingabe der Operanden die Division auszuführen. Das Ganze können Sie noch mit ein paar erklärenden *Labeln* versehen, und dann müssten Sie ungefähr die Oberfläche haben, die in Abbildung 4.6 zu sehen ist.

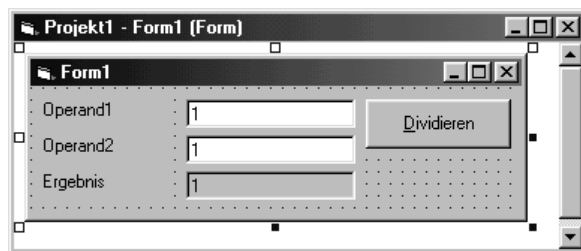


Abbildung 4.6:
Oberfläche des
Testprogramms
»Division durch 0«

Damit das Programm mit Leben erfüllt wird, wird jetzt die Ereignisroutine *Click* des Commandbuttons mit folgenden Programmzeilen gefüllt:

```
Private Sub BT_Dividieren_Click()
Dim db1Operand1 As Double
Dim db1Operand2 As Double
db1Operand1 = Val(TX_Operand1.Text)
db1Operand2 = Val(TX_Operand2.Text)
LB_Ergebnis = db1Operand1 / db1Operand2
End Sub
```

4 Workshop: Steuerung des Programmablaufs

Bei der obigen Programmsource wurde bewusst die *If...Then*-Struktur zur Abfrage der Division durch 0 weggelassen. Wenn Sie das Programm jetzt starten und für Operand 2 eine 0 eingeben, erhalten Sie die Fehlermeldung aus Abbildung 4.7.

Abbildung 4.7:
Laufzeitfehler
Division durch Null



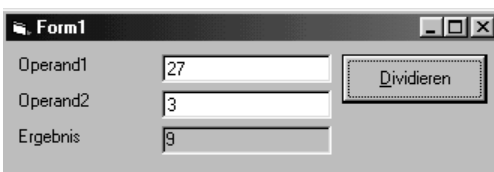
Laufzeitfehler!

Da es sich um einen Laufzeitfehler handelt, wird Ihr Programm nach Ausgabe der Fehlermeldung beendet. Genau dies wird jetzt durch eine gezielte Abfrage verhindert. Die notwendige Erweiterung sehen Sie in den folgenden Programmzeilen:

```
Private Sub BT_Dividieren_Click()  
Dim db1Operand1 As Double  
Dim db1Operand2 As Double  
db1Operand1 = Val(TX_Operand1.Text)  
db1Operand2 = Val(TX_Operand2.Text)  
If db1Operand2 = 0 Then  
    LB_Ergebnis = "Division durch 0 !"  
Else  
    LB_Ergebnis = db1Operand1 / db1Operand2  
End If  
End Sub
```

Der in diesem Zusammenhang für uns interessante Teil befindet sich in der Zeile, die mit *If* anfängt. Im Vergleich wird die Variable, die den *Operand 2* enthält, mit 0 verglichen, d.h. die Division wird nur ausgeführt, wenn *Operand 2* nicht 0 ist.

Abbildung 4.8:
Normale Division



In Abbildung 4.8 können Sie sehen, dass unser Programm eine normale Division durchführt und das Ergebnis auch korrekt ist.



Abbildung 4.9:
Keine Division

Abbildung 4.9 zeigt Ihnen den Versuch, eine Division durch 0 durchzuführen. Das Programm verhindert in diesem Fall, dass die Division durchgeführt wird, und gibt stattdessen einen Warnhinweis als Ergebnis aus. Würde die Division durch 0 tatsächlich ausgeführt werden, wäre ein Laufzeitfehler und damit das Ende des Programms die Folge.

**Warnhinweis statt
Laufzeitfehler**

4.1.4 Die Select-Case-Struktur

Die *Select-Case*-Struktur ist der *If...Then...Elseif*-Anweisung sehr ähnlich. Wenn die Bedingungen einer *If...Then...Elseif*-Anweisung sich immer auf einen Testausdruck beschränken, so kann sie durch eine *Select-Case*-Struktur ersetzt werden.

```
Select Case Testausdruck
[Case Ausdrucks]liste
[Anweisungen] ...
.
.
.
[Case Else
elseAnw]
End Select
```

Syntax

Die Anweisung beginnt mit den Schlüsselworten *Select Case*. Danach wird der Testausdruck angegeben, auf den sich alle folgenden Fälle beziehen. Innerhalb der Struktur werden mit dem Schlüsselwort *Case* Ausdruckslisten gegen den Testausdruck geprüft. Ergibt diese Prüfung den Zustand *wahr*, so wird der zugehörige Anweisungsblock ausgeführt und anschließend das Programm nach der *End Select*-Anweisung fortgeführt.

Die Ausdruckslisten können eine vielfältige Form haben. Im einfachsten Fall wird genau ein Wert gegen den Testausdruck geprüft. Die *Case*-Anweisung lautet dann:

```
Case 1
Case "a"
```

Soll auf mehrere Ausdrücke geprüft werden, werden die einzelnen Werte durch Komma getrennt angegeben:

```
Case 1,2,3
```

```
Case "a", "A"
```

Soll auf Werte geprüft werden, die sich in einer Reihenfolge befinden, so kann durch das Schlüsselwort *To* ein Bereich angegeben werden.

```
Case 1 to 9
```

```
Case "a" to "z"
```

Durch die Verwendung des Schlüsselwortes *Is* können sogar Vergleichsoperatoren in die Ausdrucksliste einbezogen werden:

```
Case Is < 100
```

```
Case Is > "a"
```



4.1.5 Übung: Verwenden der Select...Case-Struktur

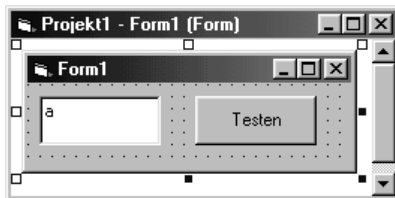
Schreiben Sie ein Programm, welches einen eingegebenen Buchstaben prüft und ausgibt, ob es sich um einen Kleinbuchstaben, Großbuchstaben oder eine Zahl handelt. Geben Sie auch eine Meldung aus, wenn keine der genannten Bedingungen zutrifft.

Lösung

Diese Übung wird Ihnen sicherlich bekannt vorkommen. Es handelt sich nämlich um die dritte Teilübung unter »Verwendung der If...Then-Struktur«. Da die *Select...Case*-Anweisung im Prinzip eine Spezialisierung der *If...Then*-Anweisung darstellt, liegt es nahe, den Versuch zu machen die gleiche Aufgabe mit den zwei unterschiedlichen Entscheidungsstrukturen zu lösen.

Zur Lösung der Aufgabe benötigen wir zunächst eine Oberfläche, die die Eingabe eines Zeichen erlaubt, und eine Schaltfläche, die den Test anstößt. Das Formular sollte also eine *TextBox* und einen *CommandButton* beinhalten ().

Abbildung 4.10:
Oberfläche Übung
Select...Case



Um die Aufgabe zu lösen, muss der eingegebene Buchstabe geprüft werden. Es handelt sich bei allen drei Prüfungen um Bereiche, so dass jeweils das Schlüsselwort *To* mit Anfangs- und Endwert verwendet werden kann. Die notwendigen Programmzeilen sehen Sie in folgendem Programmauszug:


```

Private Sub BT_Aufgabe_Click()
Select Case TX_Eingabe.Text
  Case "A" To "Z"
    MsgBox TX_Eingabe.Text & " ist ein Großbuchstabe."
  Case "a" To "z"
    MsgBox TX_Eingabe.Text & " ist ein Kleinbuchstabe."
  Case "0" To "9"
    MsgBox TX_Eingabe.Text & " ist eine Zahl."
  Case Else
    MsgBox TX_Eingabe.Text & " ist weder ein Großbuchstabe noch ein
Kleinbuchstabe noch eine Zahl."
End Select
End Sub

```

Der resultierende Programmcode ist sehr übersichtlich. Bei der Lösung mit der *If...Then*-Anweisung sind die Bedingungen deutlich länger. Für obige Übung wäre also die *Select...Case*-Anweisung vorzuziehen. Da jedoch beide Lösungen die Aufgabe korrekt erledigen, ist es eher eine Geschmacksfrage, welche der beiden Lösungen gewählt wird.

Wenn Sie das Programm starten und den Defaultwert a in der TextBox belassen, so erhalten Sie als Ergebnis korrekt den Dialog aus Abbildung 4.11.



Abbildung 4.11:
Prüfung des
Defaultwertes a

Testen Sie jede Anwendung auf jeden Fall auf Herz und Nieren. Gerade bei Bereichsprüfungen sollten zumindest alle Grenzwerte getestet werden. Das heißt, Sie sollten den Test bei dieser Übung mit a und z, A und Z, 0 und 9 und wenigstens einem Wert durchführen, der in den *Case Else*-Zweig führt (Abbildung 4.12).



Abbildung 4.12:
Auch der *Case Else*-
Zweig sollte
getestet werden

4.1.6 Die GoTo-Anweisung

Sprünge Eine einfache Möglichkeit, den sequenziellen Programmablauf zu beeinflussen, ist die Verwendung von Sprungmarken und der Anweisung *GoTo*. Wie der Name der Anweisung schon sagt, wird dabei an eine vordefinierte Stelle, nämlich die Sprungmarke, gesprungen.

Syntax Sprungmarke:

```
.  
GoTo Sprungmarke
```

Eine Sprungmarke muss folgenden Regeln entsprechen:

- ▶ Sie beginnt in der ersten Spalte.
- ▶ Sie endet mit einem Doppelpunkt.
- ▶ Sie muss in der Prozedur, in der sie verwendet wird, einen eindeutigen Namen haben.

Die *GoTo*-Anweisung selbst besteht aus zwei Teilen, dem Schlüsselwort und der Sprungmarke, zu der gesprungen werden soll. Dabei ist zu beachten, dass der Doppelpunkt nicht dem Namen der Sprungmarke zugehörig ist, also bei der Verwendung in der *GoTo*-Zeile weggelassen wird.

prozedurlokal Eine *GoTo*-Anweisung kann nur Sprungmarken erreichen, die innerhalb der eigenen Prozedur liegen.

Die *GoTo*-Anweisung hat im Gegensatz zu den bisher vorgestellten Kontrollstrukturen keine feste Form. Wenn in Ihrer Prozedur ein *GoTo*-Aufruf stattfindet, so können Sie nicht wissen, ob sich die Sprungmarke oberhalb oder unterhalb der derzeitigen Eingabemarke befindet.

Dies mag bei einer kleinen Prozedur, die im Editor auf einen Bildschirm passt, noch kein Problem sein, aber wenn die Prozedur größer wird und womöglich mehrere *GoTo*'s mit vielen Sprungmarken definiert werden, werden Sie sehr schnell den Überblick verlieren.

Funktioniert ein solcher Programmteil nicht korrekt, ist es sehr aufwändig, den Fehler zu lokalisieren, da meist nicht klar erkennbar ist, welchen Weg das Programm denn nun eigentlich nimmt.

Daher ist es besser, die *GoTo*-Anweisung so wenig wie möglich zu verwenden.

In einem Fall kann die *GoTo*-Anweisung allerdings sinnvoll verwendet werden. Wenn Sie zur Entwicklungszeit nur einen Teil einer Prozedur testen wollen, können Sie die *GoTo*-Anweisung verwenden, um den nicht auszuführenden Teil der Prozedur temporär zu deaktivieren. Sind Sie mit Ihren Tests fertig, wird die *GoTo*-Anweisung wieder entfernt.

Das folgende Beispiel zeigt die Anwendung der *GoTo*-Anweisung:

```
GoTo weiter
For i = 126 To 32 Step -1
    List1.AddItem Chr(i)
    If Chr(i) = "z" Then List1.AddItem "Kleinbuchstabe z gefunden an
Stelle " & i: Exit For Else List1.AddItem "istkein kleines z"
Next i
weiter:
```



In obigem Beispiel wird die komplette *For...Next*-Schleife übersprungen. Vergessen Sie aber nicht, die *GoTo*-Anweisung nach dem Test wieder zu entfernen. Die *For...Next*-Schleife wird sonst nie ausgeführt.

4.1.7 Fehlerbehandlung: On Error GoTo

Ein wirklich sinnvoller Einsatz der *GoTo*-Anweisung ist im Zusammenhang mit einer Fehlerbehandlung (engl. Error Handling) gegeben. Dabei tritt das Schlüsselwort *GoTo* allerdings nicht einzeln auf, sondern im Zusammenhang mit den Schlüsselwörtern *On Error*.

Tritt in einem Programm ein Laufzeitfehler auf, so wird normalerweise eine Visual Basic-Standardmeldung ausgegeben, die Fehlernummer und einen beschreibenden Text beinhaltet. Sobald diese Fehlermeldung vom Anwender bestätigt wurde, wird das Programm beendet. Im Grunde handelt es sich hierbei um einen sanften Absturz des Programms.

Mit einer Fehlerbehandlungsroutine wird ein Laufzeitfehler innerhalb einer Prozedur abgefangen. Der normale Programmablauf wird unterbrochen und in der Fehlerbehandlungsroutine weitergeführt. Dort können Programmanweisungen stehen, die den aufgetretenen Fehler analysieren, dokumentieren und gegebenenfalls beheben. Ihr Programm läuft nach Abarbeitung der Fehlerbehandlungsroutine jedoch weiter.

**Fehlerbehand-
lungsroutine fängt
Laufzeitfehler ab**

Eine Fehlerbehandlungsroutine kann über drei unterschiedliche Möglichkeiten aktiviert werden:

```
On Error GoTo Sprungmarke
On Error Resume Next
On Error GoTo 0
```

Syntax

Die erste Anweisung erlaubt bei Auftreten eines Laufzeitfehlers den Sprung zu einer definierten Sprungmarke innerhalb derselben Prozedur. Nach der Sprungmarke wird die eigentliche Fehlerbehandlung programmiert. Die Prozedur kann dort regulär beendet werden. Es ist auch möglich, durch Aufruf der Anweisung *Resume* die Programmabarbeitung an der Stelle fortzusetzen, die den Fehler verursacht hat. Dies macht natürlich nur dann Sinn, wenn in der Fehlerbehandlungsroutine die Ursache für das Auftreten des Laufzeitfehlers behoben wurde.

Mit der zweiten Anweisung wird eine Fehlerbehandlungsroutine aktiviert, die einen Laufzeitfehler »ignoriert«. D.h. tritt nach dieser Anweisung ein Laufzeitfehler auf, wird das Programm mit der folgenden Programmanweisung fortgesetzt.

Normalerweise wird diese Art der Fehlerbehandlung innerhalb einer Prozedur nur für spezielle Programmanweisungen aktiviert und direkt danach wieder deaktiviert. Die Deaktivierung einer Fehlerbehandlungsroutine erfolgt durch die dritte Anweisung.

Tritt innerhalb einer Fehlerbehandlungsroutine ein weiterer Laufzeitfehler auf, so wird dieser an die aufrufende Prozedur weitergereicht. Wenn dort eine Fehlerbehandlung aktiviert ist, wird der aufgetretene Fehler dort verarbeitet.



4.1.8 Übung: Schreiben einer Fehlerbehandlungsroutine

Schreiben Sie ein Programm, welches zwei eingegebene Zahlen miteinander multipliziert. Verwenden Sie für die internen Variablen den Datentyp *Integer*. Sorgen Sie durch eine Fehlerbehandlungsroutine dafür, dass bei Zahlen oder Ergebnissen, die den Wertebereich des Datentyps *Integer* überschreiten, kein Laufzeitfehler auftritt, der das Programm beendet.

Lösung

Um die Übung zu lösen, benötigen wir eine Oberfläche, die es erlaubt, zwei Zahlen einzugeben, ein Ergebnis auszugeben und die Multiplikation auszulösen. Abbildung 4.13 zeigt die Oberfläche meiner Lösung.

Abbildung 4.13:
Oberfläche des
Programms
»Fehlerbehand-
lungsroutine«

Der Programmcode wird in der Ereignisroutine Click der Schaltfläche hinterlegt. Um die Funktion einer Fehlerbehandlung zu demonstrieren, wird zunächst der Programmcode ohne Fehlerbehandlung eingegeben. Er sieht dann wie folgt aus:

```
Private Sub BT_Multiplizieren_Click()  
Dim intZahl1 As Integer  
Dim intZahl2 As Integer
```

```

Dim intErgebnis As Integer
intZahl1 = CInt(TX_Zahl1.Text)
intZahl2 = CInt(TX_Zahl2.Text)
intErgebnis = intZahl1 * intZahl2
TX_Ergebnis.Text = intErgebnis
End Sub

```

Im ersten Schritt werden drei Variablen des Typs *Integer* definiert. Dieser Datentyp wurde gewählt, da durch den engen Wertebereich, vor allem beim Ergebnis der Multiplikation, die Bereichsüberschreitung schon bei relativ kleinen Zahlen auftritt.

Nachfolgend werden die aktuellen eingegebenen Werte in den deklarierten Variablen gespeichert. Mit der Funktion *CInt* werden dabei die eingegebenen Zeichenketten in eine Variable des Datentyps *Integer* konvertiert.

In den beiden letzten Zeilen des Programmcodes wird schließlich die Berechnung durchgeführt und anschließend das Ergebnis in der TextBox *TX_Ergebnis* ausgegeben.

Kritisch ist an diesem Programm die Benutzereingabe, denn es wurde nichts programmiert, was die Benutzereingabe auf gültige Werte beschränkt. Das heißt, durch eine falsche Benutzereingabe könnten folgende Fehler auftreten:

- Die Benutzereingabe ist nicht als Zahl interpretierbar (Abbildung 4.14).

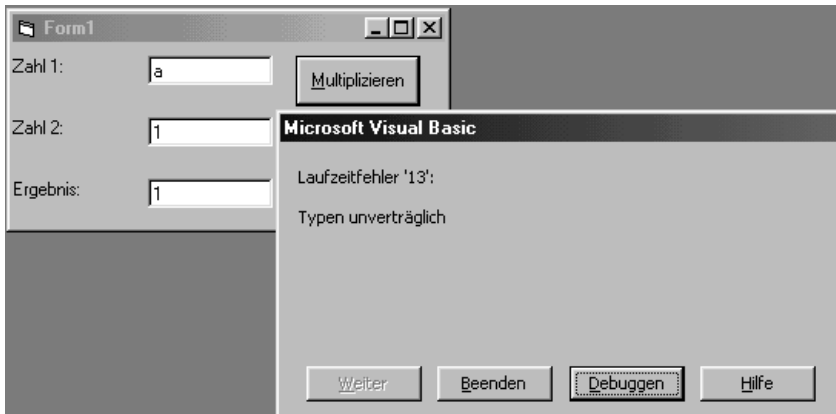


Abbildung 4.14:
Laufzeitfehler
»Typen
unverträglich«

- Die Benutzereingabe ist eine Zahl, liegt aber über dem Wertebereich des Datentyps *Integers* (Abbildung 4.15).
- Die Benutzereingaben sind Zahlen im Wertebereich des Datentyps *Integer*, aber das Ergebnis der Multiplikation liegt über dessen Wertebereich (Abbildung 4.16).

Abbildung 4.15:
Laufzeitfehler
»Überlauf«

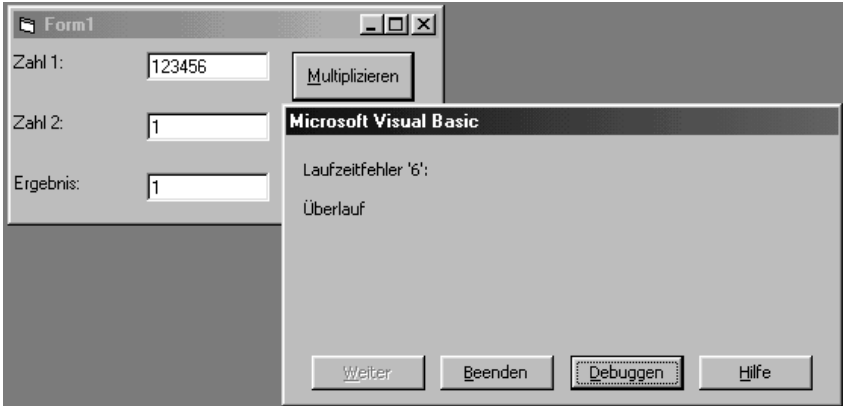
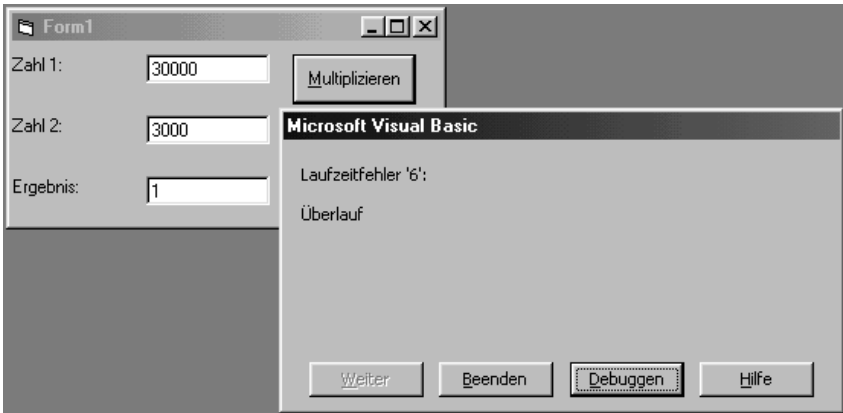


Abbildung 4.16:
Laufzeitfehler
»Überlauf« bei
Berechnung



Das Ergebnis dieser Fehleingaben ist in jedem Fall ein Laufzeitfehler, der das Programm beendet. Der Laufzeitfehler *Überlauf* tritt im ersten Fall bereits bei der Konvertierung der Eingabe, im zweiten Fall erst bei der Berechnung auf.

Um diese Fehler programmtechnisch abzufangen, müssten die Eingaben auf den Wertebereich überprüft werden. Dies wäre sicherlich noch recht einfach zu bewerkstelligen, aber das Ergebnis der Multiplikation kann nicht vor der Rechnung, die den Laufzeitfehler auslöst, geprüft werden.

Daher muss dieses Programm mit einer Fehlerbehandlungsroutine ausgestattet werden, wie im folgenden Programmcode zu sehen:

```
Private Sub BT_Multiplizieren_Click()  
Dim intZahl1 As Integer  
Dim intZahl2 As Integer  
Dim intErgebnis As Integer  
On Error GoTo BT_Multiplizieren_Click_Err  
intZahl1 = CInt(TX_Zahl1.Text)
```

```

intZahl2 = CInt(TX_Zahl2.Text)
intErgebnis = intZahl1 * intZahl2
TX_Ergebnis.Text = intErgebnis
Exit Sub
BT_Multiplizieren_Click_Err:
    TX_Ergebnis.Text = "Überlauf"
End Sub

```

Zunächst wird als erste Programmzeile nach den Variablendeklarationen mit der *On Error GoTo*-Anweisung die Fehlerbehandlungsroutine aktiviert. Die hier angegebene Sprungmarke muss in das Programm eingefügt werden. Üblicherweise wird eine Fehlerbehandlungsroutine an das Ende des normalen Funktionsablaufes angehängt.

Dann werden nach der Sprungmarke die Anweisungszeilen eingefügt, die den Fehler »behandeln«. In unserem Fall wird lediglich eine informative Meldung ausgegeben. Hierzu wird die TextBox *TX_Ergebnis* verwendet.

Ein sehr wichtiger Punkt ist die Programmzeile, die vor der Sprungmarke steht: `Exit Sub`. Fehlt diese Anweisung, so wird bei jeder Berechnung als Ergebnis »Überlauf« ausgegeben, da die Sprungmarke nicht automatisch von Visual Basic als Ende des normalen Prozedurablaufs interpretiert wird.



Wenn Sie das Programm jetzt starten, werden Sie feststellen, dass die Laufzeitfehler nicht mehr auftreten. Die Falscheingaben werden jetzt abgefangen und mit der Meldung »Überlauf« im Ergebnis-Textfeld quittiert (Abbildung 4.17). Das Programm läuft weiter und der Anwender kann seine Eingaben korrigieren.

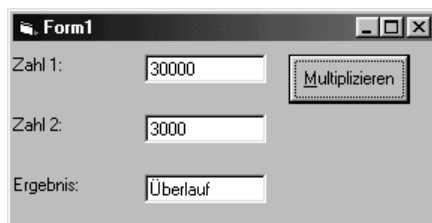


Abbildung 4.17: Laufzeitfehler »Überlauf« wurde abgefangen

Allerdings werden momentan alle Laufzeitfehler über eine Routine gemacht, in der nicht unterschieden wird, ob der Fehler bei der Konvertierung einer Zahl oder bei der Berechnung auftrat. Bei Eingabe eines kleinen *a* wird, wie Sie in Abbildung 4.18 sehen, ebenfalls die Überlauf-Meldung generiert.

Dieses Problem könnte durch die Abfrage des aufgetretenen Fehlers über das *Error*-Objekt gelöst werden. Die notwendigen Programmzeilen sehen Sie in folgendem Programmauszug:

```

BT_Multiplizieren_Err:
    If Err.Number = 13 Then ' Typen unverträglich

```

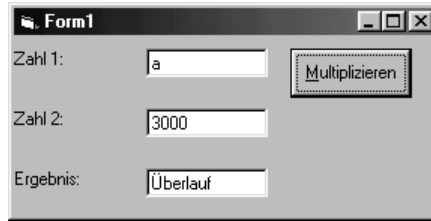


Abbildung 4.18:
Überlauf?

```

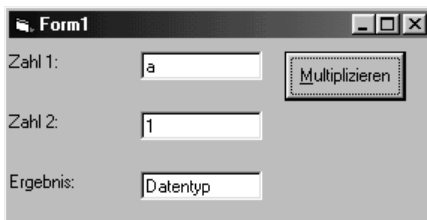
    TX_Ergebnis.Text = "Datentyp"
Else
    TX_Ergebnis.Text = "Überlauf"
End If

```

In der Fehlerbehandlungsroutine wird die aufgetretene Fehlernummer über `Err.Number` ermittelt und mit der Fehlernummer 13 verglichen. Fehlernummer 13 entspricht dem Fehler »Typen unverträglich« aus Abbildung 4.14. Abhängig von dieser Prüfung wird jetzt eine differenzierte Fehlermeldung ausgegeben.

Wird jetzt ein kleines a eingegeben, wird die Ausgabe »Datentyp« im Ergebnis-Textfeld angezeigt (Abbildung 4.19).

Abbildung 4.19:
Spezielle Fehler-
meldung für
ungültigen
Datentyp



differenzierte Fehlermeldungen durch mehrere Fehlerbehand- lungsroutinen

Eine weitere Möglichkeit, differenzierte Fehlermeldungen auszugeben, besteht darin, mehrere Fehlerbehandlungsroutinen zu schreiben und diese im Programmcode wechselweise zu aktivieren und zu deaktivieren. Mit dieser Methodik haben Sie die Möglichkeit festzustellen, an welcher Stelle in Ihrer Prozedur der Fehler aufgetreten ist. Sie können dann beispielsweise unterscheiden, welche der beiden Eingaben den Fehler verursacht.

Ein entsprechend abgeänderter Programmcode würde wie folgt aussehen:

```

Private Sub BT_Multiplizieren_Click()
    Dim intZahl1 As Integer
    Dim intZahl2 As Integer
    Dim intErgebnis As Integer
    On Error GoTo Eingabe1_err
    intZahl1 = CInt(TX_Zahl1.Text)
    On Error GoTo 0
    On Error GoTo Eingabe2_err
    intZahl2 = CInt(TX_Zahl2.Text)

```



```

On Error GoTo 0
On Error GoTo BT_Multiplizieren_Err
intErgebnis = intZahl1 * intZahl2
TX_Ergebnis.Text = intErgebnis
Exit Sub
Eingabe1_err:
    If Err.Number = 13 Then ' Typen unverträglich
        TX_Zahl1.Text = "Datentyp"
    Else
        TX_Zahl1.Text = "Überlauf"
    End If
    Exit Sub
Eingabe2_err:
    If Err.Number = 13 Then ' Typen unverträglich
        TX_Zahl2.Text = "Datentyp"
    Else
        TX_Zahl2.Text = "Überlauf"
    End If
    Exit Sub
BT_Multiplizieren_Err:
    TX_Ergebnis.Text = "Überlauf"
End Sub

```

Für die Konvertierung der Zahl wird jetzt jeweils eine eigene Fehlerbehandlungsroutine aktiviert. Nach erfolgreicher Konvertierung wird dies über die Anweisung `On Error GoTo 0` wieder deaktiviert.

In der Fehlerbehandlungsroutine wird der Fehler anhand des *Err*-Objekts untersucht und eine differenzierte Fehlermeldung in der TextBox ausgegeben, die den Fehler verursacht hat. Danach steht die *Exit Sub*-Anweisung, damit die folgenden Anweisungen der nächsten Sprungmarke nicht ausgeführt werden.

Als Ergebnis dieser Programmierarbeit erhalten Sie ein Programm, welches fehlerfrei läuft und dem Anwender durch eine Ausgabe einen Hinweis gibt, welcher Art der aufgetretene Fehler ist. Somit hat er die Möglichkeit seine Eingabe ohne weitere Untersuchung an der richtigen Stelle zu korrigieren und die Berechnung dann zu wiederholen.

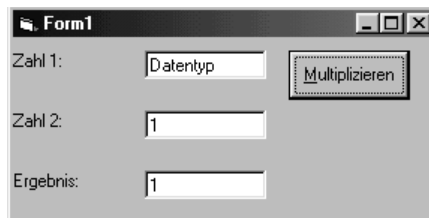


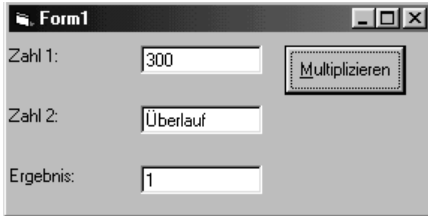
Abbildung 4.20:
Fehlermeldung bei
Eingabe kleines a
für Zahl 1

Abbildung 4.20 zeigt die Ausgabe bei Eingabe des Kleinbuchstabens a in der TextBox für Zahl 1. Wie Sie sehen, wird über das *Err*-Objekt ermittelt, dass der

4 Workshop: Steuerung des Programmablaufs

Datentyp nicht verträglich ist. Zudem wird die Ausgabe der Fehlermeldung in der TextBox für Zahl 1 gemacht. Somit kann der Anwender sofort auch dort seine korrigierte Zahl eingeben.

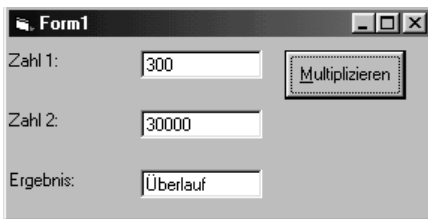
Abbildung 4.21:
Fehlermeldung bei
Eingabe 35000
für Zahl 2



The screenshot shows a window titled 'Form1' with three text boxes and a button. The first text box is labeled 'Zahl 1:' and contains the number '300'. The second text box is labeled 'Zahl 2:' and contains the text 'Überlauf'. The third text box is labeled 'Ergebnis:' and contains the number '1'. To the right of the 'Zahl 1' and 'Zahl 2' boxes is a button labeled 'Multiplizieren'.

In Abbildung 4.21 wird getestet, ob die Fehlerbehandlung für die zweite Zahl ebenfalls funktioniert. Eingegeben wurde die Zahl 35000. Dies führt bei der Konvertierung und Zuweisung zum bekannten Überlauf, der in der Fehlerbehandlungsroutine auch korrekt erkannt und in der richtigen TextBox ausgegeben wird.

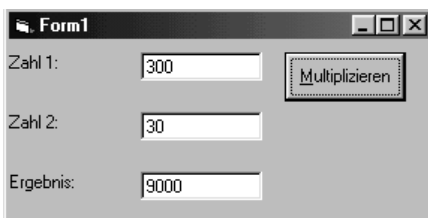
Abbildung 4.22:
Fehlermeldung,
weil Multiplikation
Überlauf erzeugt



The screenshot shows a window titled 'Form1' with three text boxes and a button. The first text box is labeled 'Zahl 1:' and contains the number '300'. The second text box is labeled 'Zahl 2:' and contains the number '30000'. The third text box is labeled 'Ergebnis:' and contains the text 'Überlauf'. To the right of the 'Zahl 1' and 'Zahl 2' boxes is a button labeled 'Multiplizieren'.

Der dritte Test in Abbildung 4.22 betrifft einen Überlauf bei der Multiplikation. Der Fehler wird korrekt erkannt und in der richtigen Textbox ausgegeben.

Abbildung 4.23:
Gültige Eingaben
und korrektes
Ergebnis



The screenshot shows a window titled 'Form1' with three text boxes and a button. The first text box is labeled 'Zahl 1:' and contains the number '300'. The second text box is labeled 'Zahl 2:' and contains the number '30'. The third text box is labeled 'Ergebnis:' and contains the number '9000'. To the right of the 'Zahl 1' and 'Zahl 2' boxes is a button labeled 'Multiplizieren'.

**normale Funktion
sollte auch
getestet werden**

Zu guter Letzt sollte natürlich auch nachgewiesen werden, dass das Programm funktioniert, wenn der Benutzer alle Eingaben korrekt macht. In Abbildung 4.23 ist dies der Fall. Beide eingegebenen Zahlen haben das richtige Format und den korrekten Wertebereich. Das Ergebnis ist ebenfalls korrekt und befindet sich noch innerhalb des Wertebereichs des Datentyps *Integer*.

4.2 Programmschleifen

Programmschleifen werden verwendet, wenn gleiche Anweisungen mehrfach abgearbeitet werden müssen. Jede Schleife wiederholt diese Anweisungen, bis ein *Abbruchkriterium* das Ende der Schleife verursacht.

Durch die Art des *Abbruchkriteriums* können zwei unterschiedliche Schleifentypen identifiziert werden:

- ▶ Die Anzahl Schleifendurchläufe ist bei Beginn der Schleife bekannt.
- ▶ Die Anzahl Schleifendurchläufe ist bei Beginn der Schleife unbekannt. Das Abbruchkriterium wird durch die Schleifenanweisungen oder durch ein externes Ereignis hergestellt.

Ein Beispiel für die Verwendung einer Schleife mit festgelegter Anzahl Durchläufe ist die Ausgabe einer ASCII-Tabelle. Da die Anzahl der möglichen Zeichen mit 255 festgelegt ist, kann eine Programmschleife fest mit dieser Anzahl durchlaufen und bei jedem Durchlauf ein Zeichen ausgeben.

Für diese Art der Schleifen stellt Visual Basic die Struktur der *For...Next*-Schleife zur Verfügung.

Ein sehr oft benötigtes Beispiel für die Verwendung einer Schleife mit variabler Anzahl Durchläufe ist das zeilenweise Auslesen einer Datei. Die Anzahl Zeilen innerhalb der Datei ist vor dem Auslesen normalerweise nicht bekannt. Daher wird in einer Schleife die Datei zeilenweise ausgelesen, bis ihr Ende erreicht ist. Innerhalb der Schleife wird also eine Anweisung ausgeführt (zeilenweises Lesen in der Datei), die das Abbruchkriterium letztlich herstellt.

Für diese Art von Schleifen stellt Visual Basic die Struktur der *Do...Loop*-Schleife zur Verfügung.

4.2.1 Die For...Next-Schleife

Da die *For...Next*-Schleife die Anzahl ihrer Durchläufe durch eine so genannte *Zähl-* oder *Laufvariable* kontrolliert, wird sie auch oft als Zähl- oder Laufschleife bezeichnet.

Hat die Laufvariable den festgelegten Grenzwert erreicht, wird die Schleife beendet. Der Grenzwert wird bereits beim Schleifenbeginn festgelegt.

```
For Zähler = Anfang To Ende [Step Schritt]
[Anweisungen]
[Exit For]
[Anweisungen]
Next [Zähler]
```

Grenzwerte

Syntax

In der *For*-Zeile wird die Zählvariable *Zähler* beim ersten Durchlauf auf den Wert *Anfang* initialisiert. Zudem wird in dieser Zeile festgelegt, bis zu welchem Wert (*Ende*) diese Zähl- oder Laufvariable »laufen« muss, bis ein Abbruch erfolgt.

Schrittweite Zudem kann in der ersten Zeile optional eine Schrittweite über das Schlüsselwort *Step* angegeben werden. Mit dem Wert *Schritt* wird festgelegt, welcher Wert vor dem nächsten Schleifendurchlauf zur aktuellen Laufvariable hinzuaddiert wird.

Wird das Schlüsselwort *Step* nicht verwendet, arbeitet die *For...Next*-Schleife mit einer Schrittweite von 1. Es können hier beliebige numerische Werte verwendet werden. So ist es auch erlaubt, Fließkommazahlen wie 0,1 oder negative Werte anzugeben.

negative Schrittweite Bei der Verwendung einer negativen Schrittweite muss allerdings darauf geachtet werden, dass der Anfangswert des Zählers größer sein muss als der Endwert.

Erreicht die Zählvariable den definierten Endwert, so wird die *For...Next*-Anweisung beendet und das Programm wird mit der Programmanweisung fortgesetzt, die dem Schlüsselwort *Next* folgt. Das Schlüsselwort *Next* ist also die untere Begrenzung der Schleife.



Die Angabe der Zählvariablen in der *Next*-Zeile ist optional. Wenn Sie allerdings größere Schleifen oder tiefe Verschachtelungen programmieren, erhöht sich die Lesbarkeit des Programmcodes durch die Angabe der Zählvariablen hinter dem Schlüsselwort *Next*. Sie sollten sich daher angewöhnen diese Syntax zu verwenden.

vorzeitiger Abbruch

Das Schlüsselwort *Exit For* bricht die *For...Next*-Schleife vorzeitig ab. Dabei wird das *Ende*-Kriterium nicht berücksichtigt.

Exit For wird immer dann verwendet, wenn ein außerordentlicher Zustand oder ein eingetretenes Ereignis den Abbruch der Schleife erfordert. Daher steht die *Exit For*-Anweisung immer innerhalb einer Entscheidungsstruktur, die ein zusätzliches Abbruchkriterium überprüft.



4.2.2 Übung: Verwenden der *For...Next*-Anweisung

Verwenden Sie die *For...Next*-Anweisung, um eine Listbox mit Zahlen zu füllen. Geben Sie dem Anwender die Möglichkeit die Listbox mit den Zahlen null bis zehn und zehn bis null zu füllen. Ermöglichen Sie zusätzlich das Füllen der Listbox mit elf Zahlen, wobei die Zahlen nicht fortlaufend, sondern mit einer Lücke von jeweils drei in die Listbox eingetragen werden.

Für die Lösung dieser Übung benötigen Sie folgende Funktionen des Steuerelements *ListBox*:

- ▶ `ListBox.Clear`: Löscht den aktuellen Inhalt der Listbox.
- ▶ `ListBox.AddItem` Zeichenkette: Hängt die übergebene Zeichenkette ans Ende der aktuellen Liste an.

Lösung

Zur Lösung der Übung wird eine Oberfläche benötigt, die zumindest ein Steuerelement *ListBox* enthält. Des Weiteren sollten die drei Teilübungen jeweils getrennt aufrufbar sein. Hierzu werden der Oberfläche drei Schaltflächen mit den entsprechenden Texten hinzugefügt. Meinen Vorschlag für die Oberfläche sehen Sie in Abbildung 4.24.

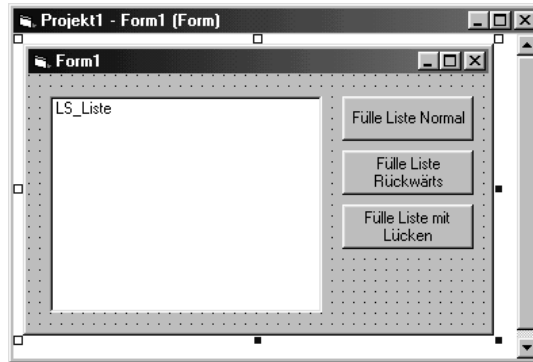


Abbildung 4.24:
Die Oberfläche des
Programms
»Verwenden der
For...Next-
Anweisung«

Der Programmcode, welcher die Listbox mit den Zahlen von 0 bis 10 füllt, wird in der Ereignisroutine der Schaltfläche mit dem Text »Fülle Liste Normal« hinterlegt. Die zur Erfüllung der Aufgabe notwendigen Programmzeilen sehen Sie im Folgenden:

```
Private Sub BT_normal_Click()
Dim i As Integer
' Löschen des Listeninhalts
LS_Liste.Clear
For i = 0 To 10
    LS_Liste.AddItem "Eintrag Nr. " & i
Next i
End Sub
```

In der ersten Programmzeile der Routine wird die Laufvariable deklariert, die später von der *For...Next*-Anweisung verwendet wird. Im Prinzip kann hier ein beliebiger Name genommen werden. Es hat sich allerdings eingebürgert, für Laufvariablen die Buchstaben i, j, k usw. zu verwenden. Dabei werden j und k nur für verschachtelte *For...Next*-Konstrukte benutzt. Wenn Sie sich bei der Namensgebung Ihrer Laufvariablen an diese Vereinbarung halten, wird auch ein fremder Entwickler es leichter haben, Ihren Programmcode zu lesen.



Nach der Deklaration der Laufvariablen wird zunächst der aktuelle Inhalt der Liste gelöscht. Direkt nach dem Programmstart ist die Liste zwar leer, aber da Ihr Programm später noch um zwei weitere Funktionen erweitert wird, die die Liste füllen, sollte bereits jetzt dafür gesorgt werden, dass die Liste beim Programmstart auf jeden Fall leer ist, bevor sie gefüllt wird.

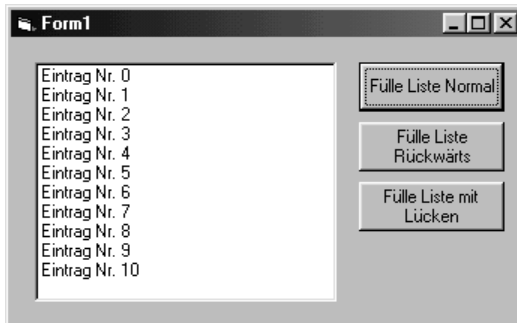
**Listeninhalt
löschen nicht
vergessen**

Im letzten Teil der Ereignisroutine wird schließlich die Liste gefüllt. Die `For...Next`-Anweisung ist ein Standardkonstrukt ohne Schrittweite. Die Laufvariable `i` wird beim Beginn der Schleife mit dem Startwert 0 initialisiert und bei jedem Durchlauf um eins erhöht.

Innerhalb der Schleife wird jeweils eine Zeichenkette an die Liste angehängt. In die Zeichenkette wird die Laufvariable `i` verkettet, so dass jeder Eintrag sich unterscheidet.

Die Ausgabe dieser Prozedur sehen Sie in Abbildung 4.25.

Abbildung 4.25:
Schleife läuft
von 0 bis 10



Um die Listbox mit den Zahlen von 10 bis 0 zu füllen, muss der Programmcode nur geringfügig geändert werden. Die Programmzeilen werden in der Ereignisroutine der Schaltfläche mit dem Text »Fülle Liste Rückwärts« hinterlegt:

```
Private Sub BT_Rückwärts_Click()  
Dim i As Integer  
' Löschen des Listeninhalts  
LS_Liste.Clear  
For i = 10 To 0 Step -1  
    LS_Liste.AddItem "Eintrag Nr. " & i  
Next i  
End Sub
```

Damit die Schleife »rückwärts« läuft, muss zunächst eine negative Schrittweite programmiert werden. Dies geschieht durch die Angabe des Schlüsselworts `Step` mit einer Schrittweite von `-1`.

Da die Laufvariable jetzt bei jedem Durchlauf um eine Zähler erniedrigt wird, müssen die Anfangs- und Endwerte der `For...Next`-Anweisung vertauscht werden. Das heißt, die Laufvariable wird mit dem Startwert 10 initialisiert und die Schleife endet, wenn die Laufvariable den Wert 0 erreicht hat.



Werden die Anfangs- und Endwerte einer `For...Next`-Schleife falsch definiert, so interpretiert Visual Basic diese Schleife, als ob bereits zu Anfang der Schleife der Endwert erreicht wäre. Es ergibt sich also keine Endlosschleife, sondern die Schleifenanweisungen werden nicht ausgeführt.

Die Ausgabe dieser Programmprozedur sehen Sie in Abbildung 4.26.

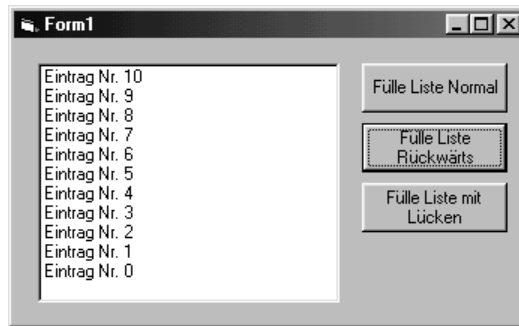


Abbildung 4.26:
Schleife läuft von
10 bis 0

Die dritte Teilübung erfordert eine neue Formulierung der *For...Next*-Zeile. Zum einen wird die Schrittweite auf drei gesetzt, damit nur jede dritte Zahl ausgegeben wird. Zum ändern müssen die Start- und Endwerte der Laufvariablen korrekt gesetzt werden. Die entsprechenden Programmzeilen werden in der Ereignisroutine der Schaltfläche hinterlegt, die den Text »Fülle Liste mit Lücken« trägt:

```
Private Sub BT_Luecken_Click()
Dim i As Integer
' Löschen des Listeninhalts
LS_Liste.Clear
For i = 0 To 31 Step 3
    LS_Liste.AddItem "Eintrag Nr. " & i
Next i
End Sub
```

Vielleicht wundern Sie sich über den Endwert, denn rein rechnerisch würde ein Endwert von 30 den Zweck genau erfüllen. In diesem Fall wurde aber zur Demonstration der Endwert 31 gewählt. Dieser wird mit einer Schrittweite von 3 nie genau erreicht, denn beim elften Durchlauf hat die Laufvariable den Wert 30, beim zwölften bereits den Wert 33.

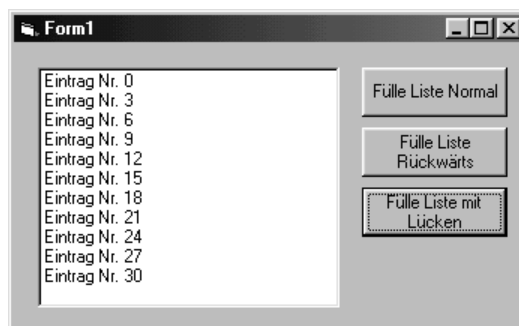


Abbildung 4.27:
10 Einträge mit
Lücken

Die Ausgabe dieser Programmanweisungen sehen Sie in Abbildung 4.27. Wie Sie sehen, wird die Aufgabe korrekt erfüllt.



Aus diesem Beispiel können Sie das genaue Verhalten der *For...Next*-Anweisung bezüglich des Endwertes ableiten. Die Prüfung des Endwertes findet vor jedem Schleifendurchlauf statt. Dabei wird der Endwert nicht auf Gleichheit, sondern auf größer gleich bei einer positiven Schrittweite und auf kleiner gleich bei einer negativen Schrittweite überprüft. Wird bei der Prüfung die Endebedingung erkannt, findet kein weiterer Schleifendurchlauf mehr statt.



4.2.3 Übung: Berechnung der Fakultät

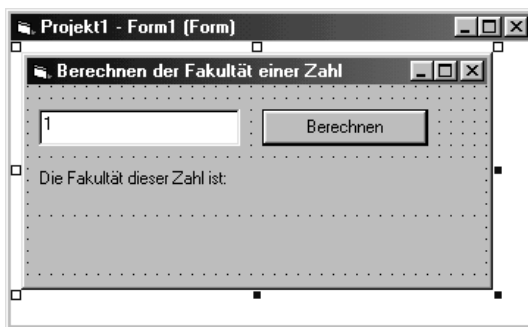
Berechnen Sie die Fakultät einer Zahl. Fangen Sie die möglichen Laufzeitfehler über Fehlerbehandlungsroutinen ab.

Lösung

Um eine Fakultät zu berechnen, wird lediglich eine Benutzereingabe gefordert. Daher benötigt die Programmoberfläche auch nur eine TextBox zur Eingabe der Zahl, deren Fakultät berechnet werden soll, eine Schaltfläche um die Berechnung anzustoßen und ein oder zwei Labels um das Ergebnis auszugeben.

Da im Programm eine Fehlerbehandlungsroutine programmiert werden soll, wird in der Oberfläche aus Abbildung 4.28 das Label so groß angelegt, dass es auch Fehlertexte beinhalten kann.

Abbildung 4.28:
Oberfläche
»Berechnung der
Fakultät«



Die Berechnung einer Fakultät wird nach folgendem Muster durchgeführt:

- ▶ Fakultät von 3 = $1 * 2 * 3$
- ▶ Fakultät von 5 = $1 * 2 * 3 * 4 * 5$

Dieses Muster ist sehr einfach in einer *For...Next*-Struktur abzubilden. Da die Fehlerbehandlungsroutinen bereits in einem früheren Abschnitt besprochen wurden, wird an dieser Stelle gleich der Programmcode inklusive der Fehlerbehandlung vorgestellt:


```

Private Sub BT_berechnen_Click()
Dim i As Double
Dim dblObergrenze As Double
Dim dblErgebnis As Double
On Error GoTo err_Convert
dblObergrenze = CDbl(Me.TX_input.Text)
On Error GoTo 0
On Error GoTo err_BT_berechnen
dblErgebnis = 1
For i = 1 To dblObergrenze
    dblErgebnis = dblErgebnis * i
Next i
LB_Ergebnis.Caption = dblErgebnis
Exit Sub
err_BT_berechnen:
    LB_Ergebnis.Caption = "außerhalb des Wertebereichs"
Exit Sub
err_Convert:
    LB_Ergebnis.Caption = "Ungültige Eingabe"
Exit Sub
End Sub

```

Da bei der mehrfachen Multiplikation sehr schnell hohe Ergebnisse zu erwarten sind, wurden alle Variablen als Datentyp *Double* deklariert. Bei der Zählvariablen *i* wäre dies nicht unbedingt notwendig, aber es soll zeigen, dass die Zählvariable nicht den Datentyp *Integer* haben muss.

Das Einlesen des Wertes erfolgt wieder über die bereits bekannte Konvertierfunktion *CDbl*, die aus der eingegebenen Zeichenkette eine Variable des Datentyps *Double* macht.

Diese Anweisungen werden in die Aktivierung und Deaktivierung der Fehlerbehandlung *err_Convert* eingekleidet. In der Fehlerbehandlung wird ausgegeben, dass ein nicht konvertierbarer Wert eingegeben wurde. Es wird nicht zwischen einem Überlauf oder einem ungültigen Datentyp unterschieden.

Für das Herzstück der Prozedur, die Berechnung in der *For...Next*-Schleife, wird eine weitere Fehlerbehandlung (*err_BT_berechnen*) aktiviert. Die eigentliche Berechnung findet in folgendem Programmstück statt:

```

dblErgebnis = 1
For i = 1 To dblObergrenze
    dblErgebnis = dblErgebnis * i
Next i

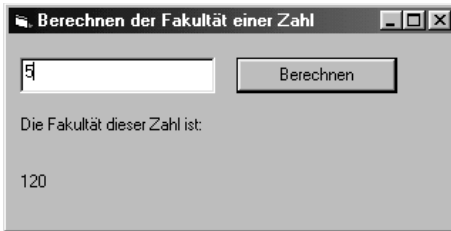
```

Um die Fakultät zu berechnen, wird mit dem Startwert 1 begonnen. Die Variable *dblErgebnis* wird vor der Schleife mit dem Startwert 1 initialisiert. Bei jedem Schleifendurchlauf wird der aktuelle Wert der Variablen mit dem aktuellen Wert der Zählvariablen multipliziert, bis die Obergrenze bzw. die Zahl erreicht ist, deren Fakultät ermittelt werden soll.

Deaktivierung der Fehlerbehandlung

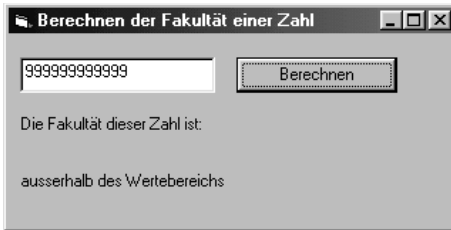
Danach wird das ermittelte Ergebnis ausgegeben. Abbildung 4.29 zeigt die Berechnung der Fakultät von 5.

Abbildung 4.29:
Berechnung der
Fakultät von 5



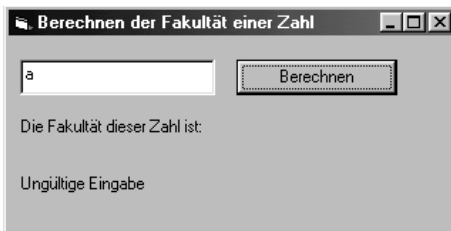
In Abbildung 4.30 sehen Sie die Ausgabe, wenn der Wertebereich des Datentyps Double bei der Berechnung überschritten wird.

Abbildung 4.30:
Überschreitung des
Wertebereichs



Des Weiteren fängt das Programm Falscheingaben des Benutzers ab, wie Sie in Abbildung 4.31 sehen.

Abbildung 4.31:
Ungültige Eingabe



4.2.4 Übung: Ausgabe einer mehrzeiligen Tabelle

Geben Sie in einer Textbox die Zahlen 0 bis 59 mehrzeilig aus. Pro Zeile sollen immer 10 Zahlen dargestellt werden.



Lösung

Die Programmoberfläche für die Lösung dieser Übung kann sehr minimalistisch gestaltet werden. Wenn Sie den Programmcode in der Ereignisroutine Load des Formulars hinterlegen, benötigen Sie keine Schaltfläche um die Prozedur aufzurufen. Das Ergebnis Ihres Programmcode wird sofort nach dem Laden des Formulars sichtbar.

Das entsprechende Formular sehen Sie in Abbildung 4.32. Es enthält lediglich eine große TextBox für die Ausgabe der Tabelle.

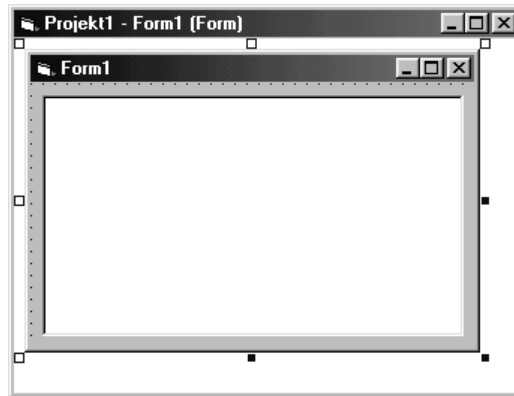


Abbildung 4.32:
Oberfläche
»Ausgabe einer
mehrzeiligen
Tabelle«

Bei der Ausgabe einer Tabelle in der geforderten Form muss jede Zeile zehn Zahlen beinhalten. D.h. eine einzelne Zeile kann über eine *For...Next*-Schleife, die insgesamt zehn Schleifendurchläufe hat, erstellt werden. Da die Tabelle jedoch mehrzeilig sein soll, muss eine weitere Schleife diese erste Schleife wiederum mehrfach aktivieren. Dies ist ein Beispiel für den Einsatz von verschachtelten *For...Next*-Schleifen.

Den kompletten Prozedurcode sehen Sie in den folgenden Zeilen.

```
Private Sub Form_Load()
    Dim i As Integer
    Dim k As Integer
    Dim ausgabe As String
    ausgabe = ""
    For i = 0 To 5
        For k = 0 To 9
            ausgabe = ausgabe & " " & Format((i * 10) + k, "00")
        Next k
        ausgabe = ausgabe & vbCrLf
    Next i
    TX_Ausgabe = ausgabe
End Sub
```

Da wir bereits festgestellt haben, dass eine verschachtelte *For...Next*-Schleifenstruktur zum Einsatz kommen soll, werden zunächst zwei Laufvariablen deklariert. Zudem wird eine Zeichenkette benötigt, die letztlich die Tabelle beinhaltet.

Vor der *For...Next*-Schleife wird diese Zeichenkette initialisiert. Danach beginnt die erste *For...Next*-Anweisung. Diese soll die notwendigen Tabellenzeilen herstellen. Gemäß der Übung muss sie also sechsmal durchlaufen werden.

verschachtelte For...Next- Schleifen

Die zweite *For...Next*-Anweisung muss eine einzelne Tabellenzeile erstellen. Da pro Zeile zehn Zahlen ausgegeben werden sollen, wird diese insgesamt zehn Durchläufe machen.

In der Tabelle sollen die Zahlen 0 bis 59 ausgegeben werden. Aus der Schleifenkonstruktion ergibt sich, dass die beiden Zählvariablen genau diesen Wertebereich abdecken, aber wie ist der Zusammenhang?

In folgender Programmzeile wird die eigentliche Arbeit erledigt:

```
ausgabe = ausgabe & " " & Format((i * 10) + k, "00")
```

An die Variable *Ausgabe* wird bei jedem Schleifendurchlauf eine neue Zahl getrennt durch ein Leerzeichen angehängt. Die Funktion *Format* sorgt lediglich dafür, dass die Zahlen zweistellig mit vorlaufenden Nullen formatiert werden. So sieht die Tabelle im Ergebnis besser aus.

Der eigentliche Trick steckt in folgender Stelle:

```
(i * 10) + k
```

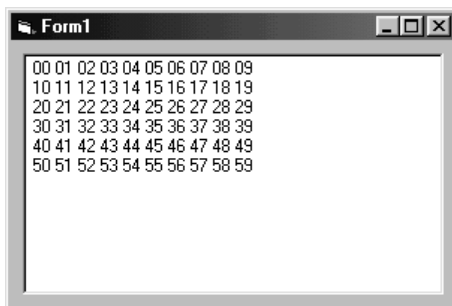
Der Schleifenzähler *i* wird als Dezimalstelle mit der Potenz 10 hoch 1 verwendet. Daraus ergibt sich bei der Berechnung für ein Dezimalsystem, dass bei jedem Durchlauf die folgende Zahl errechnet wird.

Damit jetzt die einzelnen Zeilen tatsächlich in Tabellenform angezeigt werden, muss lediglich bei jedem Durchlauf der äußeren *For...Next*-Schleife ein Zeilenvorschub an die Zeichenkette angehängt werden. Dies erfolgt mit folgender Programmzeile:

```
ausgabe = ausgabe & vbCrLf
```

Nachdem die Schleifenkonstruktion durchgelaufen ist, wird nur noch die Zeichenkette in der Textbox zur Anzeige gebracht. Nach dem Start des Programms haben Sie die Ausgabe aus Abbildung 4.33 vor sich.

Abbildung 4.33:
Ausgabe des Pro-
gramms Tabelle



4.2.5 Die Do...Loop-Schleife

Eine *Do...Loop*-Schleife wird verwendet, wenn sich die Anzahl der Schleifendurchläufe vor der Verarbeitung nicht genau bestimmen lässt.

Es werden zwei unterschiedliche Syntaxschreibweisen unterstützt.

```
Do [{While | Until} Bedingung]
  [Anweisungen]
[Exit Do]
[Anweisungen]
Loop
```

Syntax 1

```
Do
  [Anweisungen]
[Exit Do]
[Anweisungen]
Loop [{While | Until} Bedingung]
```

Syntax 2

Syntax 1 und *Syntax 2* unterscheiden sich durch den Zeitpunkt, zu welchem die Abbruchbedingung geprüft wird. *Syntax 1* prüft die Abbruchbedingung vor dem Eintritt in die Schleife. Ist die Abbruchbedingung zu diesem Zeitpunkt bereits erfüllt, werden die Anweisungen innerhalb der Schleife überhaupt nicht durchgeführt.

Eine Schleife mit *Syntax 2* würde in diesem Fall wenigstens einmalig durch die Anweisungen innerhalb der Schleife gehen, da sie erst am Ende feststellt, dass die Abbruchbedingung erfüllt ist.

Eine weiterer Unterschied ergibt sich durch das Schlüsselwort, welches der Abbruchbedingung vorangestellt ist. Das Schlüsselwort *While* legt fest, dass die Schleife durchgeführt wird, *solange* die Bedingung *wahr* ist.

While

Durch das Schlüsselwort *Until* wird die Schleife ausgeführt, *bis* die Bedingung *wahr* wird. Man könnte allerdings auch sagen, dass bei *Until* die Schleife ausgeführt wird, *solange* die Bedingung *unwahr* ist. Es handelt sich also um die gegenteilige Logik des Schlüsselworts *While*.

Until

Es gibt keinen zwingenden Grund für das Vorhandensein beider Schlüsselwörter, da durch eine entsprechende Formulierung der Abbruchbedingung jede *Until*-Schleife auch mit dem Schlüsselwort *While* abgebildet werden könnte. Allerdings ist die Formulierung dieser Bedingung manchmal mit dem einen Schlüsselwort einfacher als mit dem anderen.

Der Anfang einer *Do...Loop*-Schleife wird durch das Schlüsselwort *Do* gekennzeichnet. Das Ende der Schleife mit dem Schlüsselwort *Loop*. *Do...Loop*-Schleifen können wie die *For...Next*-Schleifen auch verschachtelt werden. Allerdings fehlt im Gegensatz zur *For...Next*-Schleife der Hinweis *Zähler* am Ende der Struktur und somit die eindeutige Zuordnung der Schleife.

Kommentare verbessern die Programmübersicht

Sie sollten deshalb gerade bei Schleifen, wie auch sonst im Programm, großzügig mit Kommentaren arbeiten. Zusätzlich sollten Sie bei verschachtelten Schleifen Einrückungen im Programmcode verwenden. Somit haben Sie beim Lesen des Programmcodes eine optische Zuordnung zur entsprechenden Schleifenebene.

Im folgenden Programmcode wurden diese Regeln zur Demonstration absichtlich missachtet:

```
Do While xKoordinate < 200
.
Anweisungen .
.
Anweisungen
Do While yKoordinate > 100
.
Anweisungen .
.
Loop
.
Anweisungen .
.
Loop
```

Schon bei diesem kleinen Stück Programmcode ist nicht ohne weiteres sofort zu erkennen, zu welchem Schleifenbeginn die entsprechenden Loop-Anweisungen gehören. Sie können sich sicherlich vorstellen, dass es schwieriger wird, den Programmcode zu lesen, je größer die Programmschleifen bzw. die Anzahl Anweisungen und die Anzahl Verschachtelungen werden.

Werden die gleichen Anweisungen mit Einrückungen versehen, ist der Programmcode, wie in folgendem Auszug, deutlich lesbarer:

```
Do While xKoordinate < 200

    Anweisungen

Do While yKoordinate > 100

    Anweisungen

Loop ' Ende der Schleife für yKoordinate

    Anweisungen
Loop ' Ende der Schleife für xKoordinate
```

Zusätzlich zu den Einrückungen wurden die Schleifenende-Anweisungen mit einem Kommentar erweitert, der einen Bezug herstellt zum entsprechenden Schleifenbeginn. Auf diese Weise können Sie auch dann eine einfache Zuordnung des Schleifenendes machen, wenn der Schleifenbeginn im Programmeditor nicht mehr sichtbar ist.

4.2.6 Übung: Verwenden der Do...Loop-Schleife



In Abschnitt 4.2.2 wurde mit der *For...Next*-Schleife eine Listbox auf unterschiedliche Arten gefüllt. Schreiben Sie dieses Programm neu und verwenden Sie statt der *For...Next*-Struktur eine *Do...Loop*-Struktur. Wenden Sie für jede Teilübung unterschiedliche Syntaxformen der *Do...Loop*-Schleife an.

Lösung

Zur Lösung der Übung wird, da ein vorhandenes Programm umgeschrieben wird, die bereits vorhandene Oberfläche (Abbildung 4.34) verwendet.

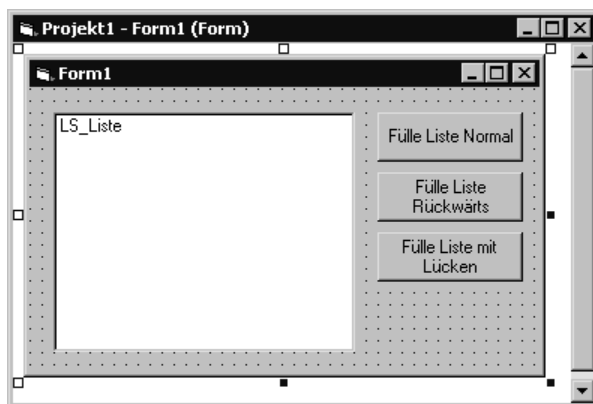


Abbildung 4.34: Oberfläche des Programms *For...Next* für *Do...Loop* verwenden

Diese Übung hat nicht nur den Zweck die *Do...Loop*-Struktur einzuüben. Sie soll auch verwendet werden, um die Verwandtschaft der *Do...Loop*- und der *For...Next*-Schleife zu demonstrieren.

Wenn Sie ein altes Projekt umschreiben, sollten Sie überlegen, ob Sie das alte Programm überschreiben oder behalten. Wenn Sie das alte Programm behalten möchten, sollten Sie die kompletten Programm- und Projektdateien in ein neues Verzeichnis kopieren. Anschließend können Sie über den Dialog DATEI->PROJEKT ÖFFNEN in der Registerkarte VORHANDEN (Abbildung 4.35) die Projektdatei des kopierten Projekts öffnen.

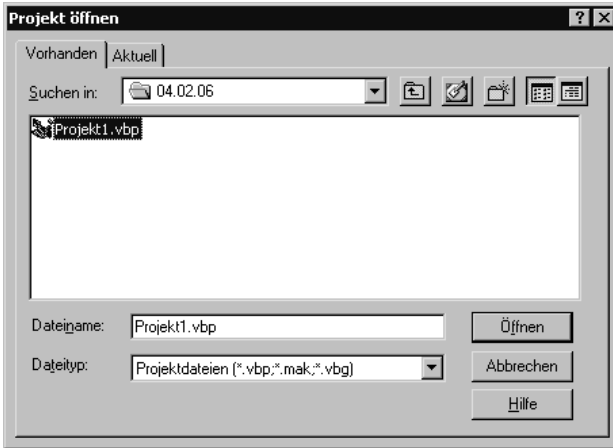


Da Visual Basic in der Projektdatei mit relativen Pfaden arbeitet, werden jetzt nur Projektdateien des neuen Verzeichnisses verwendet. Ihr altes Projekt wird von den Änderungen nicht berührt.

relative Pfade

Um die Übung zu lösen, werden Schritt für Schritt alle *For...Next*-Schleifen des vorhandenen Projektes auf die *Do...Loop*-Struktur umgestellt. Als Erstes wird die Ereignisprozedur der ersten Schaltfläche geöffnet. Die darin befindliche *For...Next*-Schleife hat folgendes Aussehen:

Abbildung 4.35:
Vorhandenes
Projekt öffnen



```
For i = 0 To 10
    LS_Liste.AddItem "Eintrag Nr. " & i
Next i
```

Es handelt sich um die einfachste Form der *For...Next*-Schleife. Die Schleife hat elf Durchläufe (von 0 bis 10), es wird keine Schrittweite definiert und der Anfangs- und Endwert liegt bereits zu Beginn der Schleife fest.

Um diese Schleife in der *Do...Loop*-Struktur abzubilden, muss vor allem die Funktion der *For*-Zeile programmiert werden. In dieser wird zuerst der Wert der Zählvariablen auf 0 initialisiert. Die entsprechende Programmzeile ist:

```
i = 0
```

Des Weiteren wird die Zählvariable bei jedem Durchlauf um eins (keine Schrittweite definiert) erhöht. Dies wird innerhalb der Schleife durchgeführt:

```
i = 0
Do
    i = i + 1
Loop While
```

Zudem muss die Überwachung der Endbedingung programmiert werden. Da wir das Schlüsselwort *While* verwenden, muss die Bedingung lauten, dass die Schleife so lange läuft, wie die Zählvariable den Wert 10 nicht überschreitet. Zudem müssen natürlich innerhalb der Schleife die gleichen Programmanweisungen ausgeführt werden. Der komplette umgestellte Programmcode hat also folgendes Aussehen:

```
i = 0
Do
    LS_Liste.AddItem "Do..Loop Eintrag Nr. " & i
    i = i + 1
Loop While i <= 10
```


Im Gegensatz zur *For...Next*-Schleife wurden zwei zusätzliche Programmzeilen zur Initialisierung der Zählvariablen und deren Iteration benötigt. Die *Do...Loop*-Schleife sollte also nur dann verwendet werden, wenn die Übung durch eine *For...Next*-Schleife nicht abgebildet werden kann.



Um einen visuellen Unterschied zum bisherigen Programm zu demonstrieren, wurde der Text um die Worte *Do...Loop* ergänzt. Das Ergebnis der Umstellung sehen Sie in Abbildung 4.36.

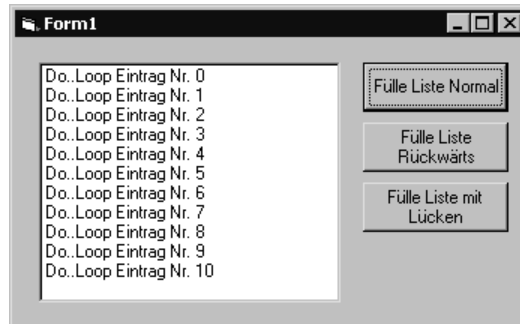


Abbildung 4.36: Ausgabe der umgestellten Ereignisprozedur

In der gleichen Art und Weise werden jetzt die beiden anderen Ereignisprozeduren des Projekts umgestellt. Die nächste Ereignisprozedur hat folgendes Aussehen:

```
Private Sub BT_Rückwärts_Click()
Dim i As Integer
' Löschen des Listeninhalts
LS_Liste.Clear
For i = 10 To 0 Step -1
    LS_Liste.AddItem "Eintrag Nr. " & i
Next i
End Sub
```

Da diese Schleife rückwärts läuft, müssen die Initialisierung der Zählvariablen, deren Iteration und die Abbruchbedingung geändert werden. Die komplette umgestellte Prozedur sieht aus wie folgt:

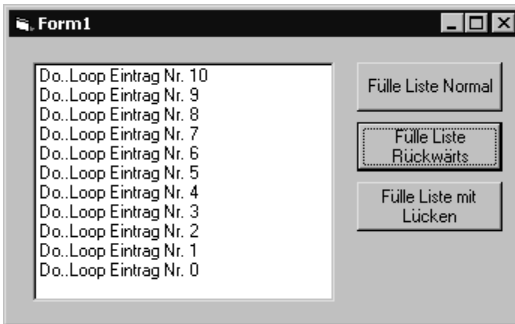
```
Private Sub BT_Rückwärts_Click()
Dim i As Integer
' Löschen des Listeninhalts
LS_Liste.Clear
i = 10
Do While i >= 0
    LS_Liste.AddItem "Do..Loop Eintrag Nr. " & i
    i = i - 1
Loop
End Sub
```

4 Workshop: Steuerung des Programmablaufs

Die Zählvariable *i* wird diesmal vor Schleifenbeginn auf den Wert 10 gesetzt. Innerhalb der Schleife wird bei jedem Durchlauf eins abgezogen. Verwendet wurde diesmal die *Do...Loop*-Schleife, deren Abbruchbedingung zu Beginn geprüft wird. Das Abbruchkriterium wird wiederum mit dem Schlüsselwort *While* geprüft. Die Schleife soll also laufen, »solange die Zählvariable größer oder gleich 0 ist«.

Die Ausgabe die Prozedur sehen Sie in Abbildung 4.37.

Abbildung 4.37:
Zählvariable läuft
rückwärts



Zu guter Letzt wird die dritte Ereignisprozedur umgestellt. Das Original sehen Sie in folgenden Programmzeilen:

```
Private Sub BT_Luecken_Click()  
Dim i As Integer  
' Löschen des Listeninhalts  
LS_Liste.Clear  
For i = 0 To 31 Step 3  
    LS_Liste.AddItem "Eintrag Nr. " & i  
Next i  
End Sub
```

Im Gegensatz zur ersten Prozedur der Übung ändert sich hier lediglich die Schrittweite, also das Maß, um welches die Zählvariable pro Durchlauf verändert wird. Da trotzdem insgesamt elf Ausgaben zu machen sind, wird die Endebedingung entsprechend formuliert. Da die Aufgabenstellung die Verwendung verschiedener *Do...Loop*-Formen verlangt, wird in diesem Fall das Schlüsselwort *Until* benutzt.

Die umgestellte Prozedur sieht wie folgt aus:

```
Private Sub BT_Luecken_Click()  
Dim i As Integer  
' Löschen des Listeninhalts  
LS_Liste.Clear  
i = 0  
Do  
    LS_Liste.AddItem "Do..Loop Eintrag Nr. " & i
```

```
i = i + 3
Loop Until i > 30
End Sub
```

Die Schleife wird diesmal ausgeführt, »bis der Schleifenzähler den Wert 30 überschreitet«. Die Ausgabe dieser Schleife ist in Abbildung 4.38 zu sehen.

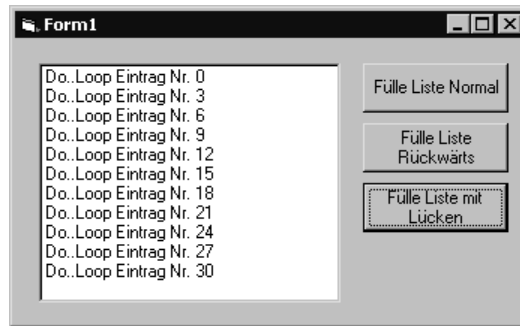


Abbildung 4.38:
Do...Loop-Schleife
mit Schlüsselwort
Until

4.2.7 Übung: Zeilenweises Einlesen einer Datei

Schreiben Sie ein Programm, welches die Datei `c:\config.sys` zeilenweise einliest und den Inhalt in einer `ListBox` darstellt.

Um diese Übung zu lösen, müssen Sie sich zunächst mit den Dateioperationen von Visual Basic beschäftigen. Eine Datei muss in Visual Basic zuerst geöffnet werden. Hierzu wird die Anweisung `Open` verwendet. Versuchen Sie die Übung alleine zu lösen, indem Sie die Visual Basic Online-Hilfe verwenden.

Hilfe zu diesem Thema erhalten Sie am einfachsten, indem Sie im Programmeditor das Schlüsselwort `Open` eingeben und dann die Taste `[F1]` drücken. Die Online-Hilfeseite (Abbildung 4.39) zur Anweisung `Open` wird automatisch geöffnet.

Die Online-Hilfe von Visual Basic ist so aufgebaut, dass Sie zu jedem Thema auch Verweise auf zugehörige Themen erhalten. Hierzu muss lediglich der Text *Siehe auch* angeklickt werden. Es öffnet sich ein Dialog (Abbildung 4.40) mit Hilfethemen, die in den Themenbereich der bereits geöffneten Hilfeseite passen.

Zusätzlich gibt es bei vielen Hilfeseiten Beispielprogramme, die sehr zum Verständnis beitragen. Falls bei einer Hilfeseite der Text *Beispiel* auswählbar ist, sollten Sie die entsprechende Hilfeseite unbedingt öffnen.

Um aus der Datei zu lesen, wird die Funktion `Line Input` verwendet. Lesen Sie zu dieser Funktion die Online-Hilfe von Visual Basic.

Nutzen Sie diese Übung, um sich mit der Visual Basic Online-Hilfe vertraut zu machen. Dieses Wissen wird Ihnen sehr oft von Nutzen sein.



Aufbau der Online-Hilfe

Line Input

4 Workshop: Steuerung des Programmablaufs

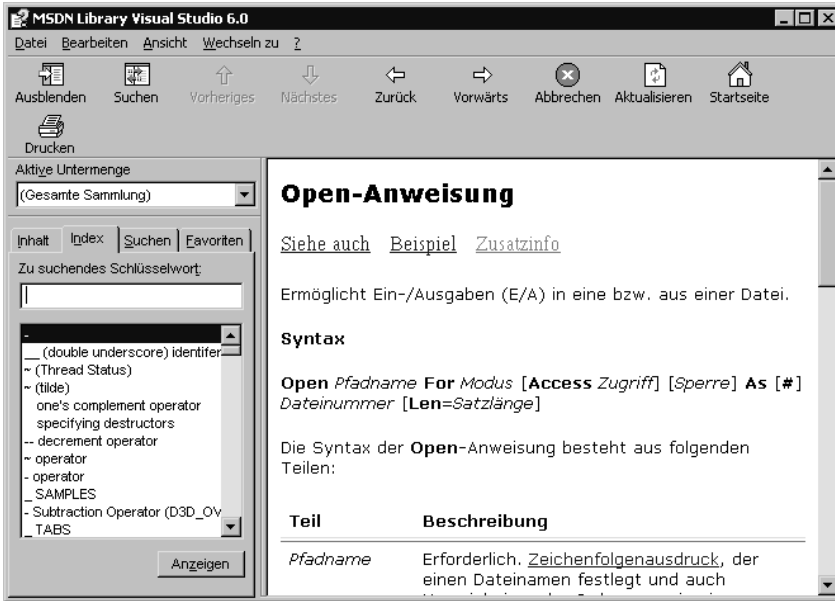


Abbildung 4.39:
Online-Hilfe der
Open-Anweisung

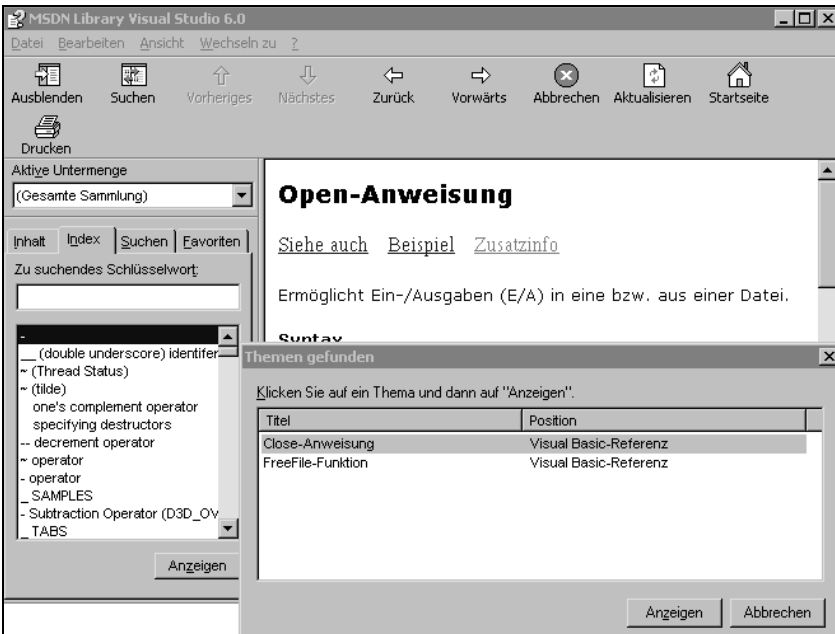


Abbildung 4.40:
Weitere themenzu-
gehörige Hilfe-
seiten

Lösung

Zur Lösung der Übung wird eine Oberfläche benötigt, die in der Lage ist den Inhalt der Datei darzustellen und die Funktion aufzurufen, welche die Datei einliest und darstellt.

Im Prinzip könnte die Darstellung des Dateiinhalts in einer Textbox erfolgen. Dazu müsste lediglich eine Variable des Datentyps String verkettet werden, wobei zwischen die einzelnen Zeilen die Konstante `vbCrLf` eingefügt wird. Es ist allerdings in diesem Fall einfacher, eine Liste zu füllen, wobei für jede Zeile der Datei ein Eintrag hinzugefügt wird.

Die Oberfläche meiner Lösung sieht also wie folgt aus:

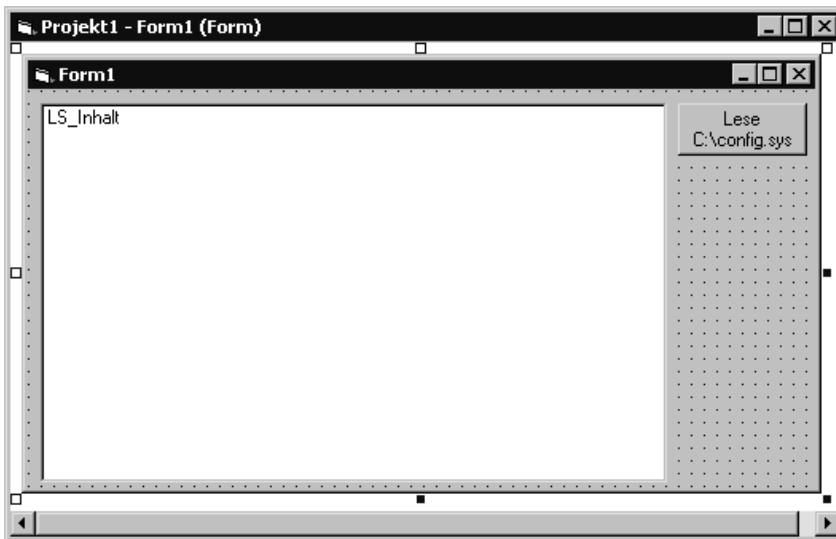


Abbildung 4.41:
Oberfläche der
Übung Datei
auslesen

Um zur eigentlichen Übung der *Do...Loop*-Schleife zu kommen, muss zunächst geklärt werden, wie eine Datei verwendet wird. Haben Sie in der Hilfe alles gefunden?

Der Name bzw. Pfad einer Datei wird nur in der *Open*-Anweisung verwendet. Alle weiteren Funktionen beziehen sich immer auf eine Dateinummer. Diese wird vom Betriebssystem verwaltet und kann bzw. sollte von diesem angefordert werden.

Für diesen Zweck wird die Funktion *FreeFile* angewendet. Sie gibt die nächste freie Dateinummer zurück, die dann in der *Open*-Anweisung verwendet wird. In den Beispielen zur Anweisung *Open* wird immer mit einer festen Nummer gearbeitet. Und tatsächlich funktioniert dies auch. Mit folgender Anweisung wird eine Datei ohne Fehlermeldung geöffnet:

FreeFile

```
Open "C:\config.sys" For Input As #1
```

Allerdings funktioniert diese Programmzeile nur dann, wenn die Dateinummer 1 nicht bereits vom System vergeben wurde. Denn das Programm kann jede Dateinummer nur einer Datei bzw. *Open*-Anweisung zuordnen.

Wurde die Dateinummer bereits verwendet, so erhalten Sie beim Öffnen der Datei mit *Open* eine Fehlermeldung (Abbildung 4.42). Sie können dies auf sehr einfache Weise produzieren, indem Sie die obige Anweisung zweimal hintereinander aufrufen:

```
Open "C:\config.sys" For Input As #1
Open "C:\config.sys" For Input As #1
```

Abbildung 4.42:
Fehlermeldung:
Datei bereits
geöffnet



Sicher wird es Ihnen nicht passieren, dass Sie den Fehler in einem normalen Programm machen, indem Sie die zwei *Open*-Anweisungen direkt hintereinander schreiben. Aber in einem Programm können durchaus mehrere Dateien gleichzeitig geöffnet sein, und diese werden dann auch meist in unterschiedlichen Prozeduren oder gar Modulen verwendet.



Es sollte Ihnen unbedingt zur Gewohnheit werden, die Dateinummer immer über den Aufruf der Funktion *FreeFile* anzufordern. Die notwendigen Programmzeilen sehen wie folgt aus:

```
Dim fnum As Integer
fnum = FreeFile
```

Nachdem eine Dateinummer angefordert wurde, kann die Datei geöffnet werden. Die entsprechende *Open*-Anweisung sehen Sie in folgender Programmzeile:

```
Open "C:\config.sys" For Input As #fnum
```

lesender Zugriff

Der Dateiname wurde in der Übung vorgegeben. Da die Datei nur gelesen wird, wird sie mit dem Schlüsselwort *For Input*, also nur für lesenden Zugriff, geöffnet.

Als Dateinummer wird die zuvor mit *FreeFile* ermittelte Variable übergeben.

Nachdem die Datei geöffnet wurde, kann aus ihr gelesen werden. Hierzu wird die Funktion *Line Input* verwendet, die pro Aufruf eine komplette Zeile der

Datei liest. Die Funktion *Line Input* hat keinen Rückgabewert, stattdessen wird die gelesene Zeile in einem Übergabeparameter abgelegt. Zudem wird der Funktion durch eine Dateinummer mitgeteilt, in welcher Datei gelesen werden soll. Die notwendigen Programmanweisungen sehen wie folgt aus:

```
Dim strZeile As String
Line Input #fnum, strZeile
```

Um die notwendige Do...Loop-Schleife zu programmieren, muss jetzt allerdings noch ein Abbruchkriterium formuliert werden. Hierzu wird eine Funktion benötigt, die ermittelt, ob das Ende der Datei erreicht ist. Diese Funktion heißt EOF. EOF ist die Abkürzung des englischen Ausdrucks »End Of File«, also Dateiende. Dieser Funktion wird eine Dateinummer übergeben. Sie liefert als Rückgabe den Wert *True*, falls das Dateiende erreicht ist.

EOF = End Of File

Die komplette Funktion sieht also wie folgt aus:

```
Private Sub BT_Readfile_Click()
Dim strZeile As String
Dim fnum As Integer
fnum = FreeFile
Open "C:\config.sys" For Input As #fnum
Do Until EOF(fnum)
    Line Input #fnum, strZeile
    LS_Inhalt.AddItem strZeile
Loop
Close #fnum
End Sub
```

Die bisher besprochenen Anweisungen wurden umgesetzt in einer *Do...Until*-Anweisung, wobei das Abbruchkriterium vor dem ersten Schleifendurchlauf geprüft wird. Das Prüfen auf das Dateiende vor dem ersten lesenden Zugriff ist sehr wichtig, weil eine Datei auch leer sein könnte. In diesem Fall ist das Dateiende sofort nach dem Öffnen der Datei erreicht. Würde bei einer leeren Datei erst am Ende der Schleife auf Dateiende geprüft, so findet ein lesender Zugriff nach dem Dateiende statt, was in einem Laufzeitfehler resultiert. Die entsprechende Fehlermeldung sehen Sie in Abbildung 4.43.

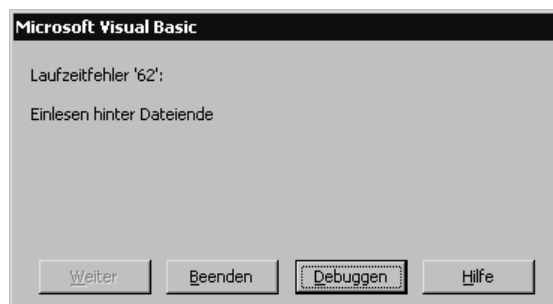


Abbildung 4.43:
Laufzeitfehler
»Lesen nach
Dateiende«

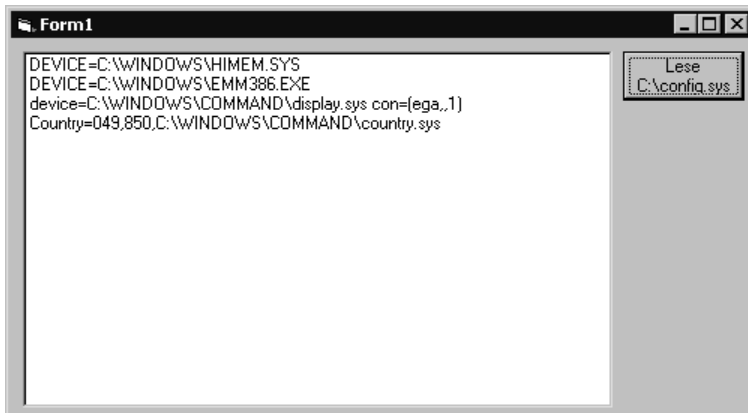
Ansonsten wurde der Programmcode um zwei weitere Anweisungen erweitert. Zum einen wurde eine Anweisung in der Schleife hinzugefügt, die die gelesene Zeile aus der Datei in einer Listbox darstellt. Zum anderen wurde die Anweisung *Close* hinzugefügt, die eine Datei schließt.



Eine geöffnete Datei sollte immer zu einem möglichst frühen Zeitpunkt auch wieder geschlossen werden. Dies ist aus zwei Gründen wichtig. Erstens wird nach dem Schließen der Datei die verwendete Dateinummer wieder freigegeben und zweitens werden erst beim Schließen der Datei alle Dateipuffer durch das Betriebssystem aufgeräumt. Dies hat bei einer Datei, in die geschrieben wurde, den Effekt, dass auch erst dann zu 100% sichergestellt ist, dass die geschriebenen Daten tatsächlich auf der Festplatte gespeichert sind.

Wenn Sie jetzt das Programm ablaufen lassen, erhalten Sie in der Listbox den Inhalt Ihrer Datei *config.sys*. Bei meinem Rechner sieht dies aus wie in Abbildung 4.44.

Abbildung 4.44:
Ausgabe des Programms Datei lesen



Mit dieser Prozedur haben Sie ein Musterprogramm, welches Sie sehr einfach für beliebige andere Dateien verwenden können. Sie müssen lediglich die Zeile mit der *Open*-Anweisung ändern und den Dateinamen austauschen.



4.2.8 Übung: Eine Warteschleife

Schreiben Sie ein Programm, welches nach dem Start einer Funktion in einer Schleife wartet, bis der Anwender eine zweite Schaltfläche klickt.

Lösung

DoEvents Haben Sie versucht diese Übung zu lösen? Nun, dann haben Sie sicher festgestellt, dass die Tücke im Detail liegt. Das Problem liegt darin, dass diese Übung nicht zu lösen ist, wenn Sie die Anweisung *DoEvents* nicht kennen.

DoEvents wird immer dann in Visual Basic benötigt, wenn eine Funktion sehr lange läuft, in dieser Zeit aber weitere Ereignisse, wie beispielsweise das Klicken einer Schaltfläche vom Programm verarbeitet werden sollen. In Visual Basic werden laufende Prozeduren nicht unterbrochen. Das heißt, wenn eine Prozedur sehr lange, z.B. in einer Endlosschleife, läuft, kann kein anderes Ereignis bearbeitet werden.

**anderen Prozessen
Zeit geben**

Für den Anwender sieht es so aus, als ob das Programm hängt, denn er drückt eine Schaltfläche, aber nichts passiert.

Mit der Anweisung *DoEvents* können Sie innerhalb einer lang laufenden Prozedur dafür sorgen, dass Visual Basic andere Ereignisse bearbeitet. Immer wenn ein *DoEvent* im Programmcode steht, wird von Visual Basic Prozesszeit abgegeben, d.h. andere anstehende Ereignisse werden abgearbeitet.

In lang laufenden Prozeduren oder Endlosschleifen sollten Sie nicht sparsam mit *DoEvents* umgehen. Gerade in Schleifen bietet es sich an, bei jedem Durchlauf ein *DoEvent* aufzurufen, damit der Rest Ihres Programms lauffähig bleibt.



Um diese Übung zu lösen, werden zumindest zwei Schaltflächen benötigt, über welche die Endlosschleife gestartet und gestoppt werden kann. Ich habe mich bei meiner Lösung dafür entschieden, in der Schleife Ausgaben auf der Oberfläche (Abbildung 4.45) zu machen, damit der Anwender sieht, dass das Programm noch läuft.

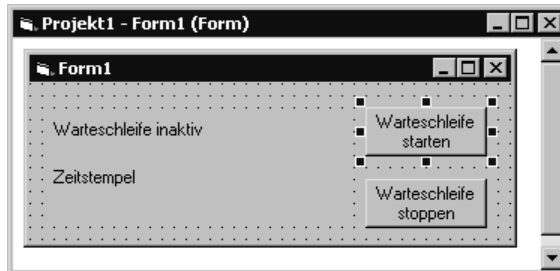


Abbildung 4.45:
Oberfläche des
Programms
Warteschleife

Die Warteschleife wird in der Ereignisprozedur der Schaltfläche *Warteschleife starten* aktiviert. Die notwendigen Programmzeilen sehen Sie im Folgenden:

```
Private Sub BT_Start_Click()
LB_Info1.Caption = "Warteschleife gestartet"
Do
    LB_info2.Caption = Format(Now, "HH:MM:SS")
    DoEvents
Loop Until False
LB_Info1.Caption = "Warteschleife angehalten um"
End Sub
```

4 Workshop: Steuerung des Programmablaufs

In der ersten und letzten Zeile der Ereignisprozedur wird jeweils ein informativer Text für den Anwender ausgegeben. Die *Do...Loop*-Schleife enthält eine weitere Ausgabe und das bereits besprochene *DoEvent*.

Die Ausgabe in der Schleife wird mit den Funktionen *Format* und *Now* gemacht. Die Funktion *Now* liefert als Rückgabewert eine Variable des Datentyps *Variant*, Untertyp *Date*, die das aktuelle Datum und die aktuelle Uhrzeit enthält.

Die Funktion *Format* formatiert diese Ausgabe in ein lesbares Format.

Die Abbruchbedingung der Schleife wurde auf *Loop Until False* gesetzt. Da der Wert *False* nie *True* werden kann, wird diese Schleife also nie abgebrochen.

Für den zweiten Teil der Übung muss aber genau dies geschehen. Wie also kann eine zweite Ereignisprozedur die Schleife über einen Abbruchwunsch informieren?

Die Lösung hierfür ist eine Modul-globale Variable. Sie wird im allgemeinen Teil des Moduls deklariert und kann somit in beiden Ereignisprozeduren verwendet werden.

Da diese Variable lediglich den Zustand Schleife aktiv/inaktiv steuern soll, genügt der Datentyp *Boolean*.

Der komplette Programmcode sieht dann aus wie folgt:

```
Option Explicit
Dim benutzerstop As Boolean
Private Sub BT_Start_Click()
    benutzerstop = False
    LB_Info1.Caption = "Warteschleife gestartet"
    Do
        LB_info2.Caption = Format(Now, "HH:MM:SS")
        DoEvents
    Loop Until benutzerstop
    LB_Info1.Caption = "Warteschleife angehalten um"
End Sub
Private Sub BT_Stop_Click()
    benutzerstop = True
End Sub
```

Mit der globalen Variablen *benutzerstop* ist die weitere Programmierung sehr einfach. Bevor in der Ereignisprozedur *BT_Start_Click* die Schleife gestartet wird, wird die Variable *benutzerstop* auf den Wert *False* gesetzt.

Die Abbruchbedingung der Schleife prüft dann, ob die Variable auf *True* gesetzt wurde. Genau dies ist die einzige Aufgabe der Ereignisprozedur *BT_Stop_Click*.

In Abbildung 4.46 sehen Sie die Programmoberfläche nach den Start der Schleife über die Schaltfläche *Schleife starten*.

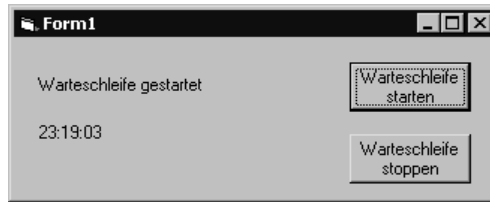


Abbildung 4.46:
Endlosschleife ist aktiv

Abbildung 4.47 zeigt die Oberfläche, nachdem die Schleife wieder gestoppt wurde.

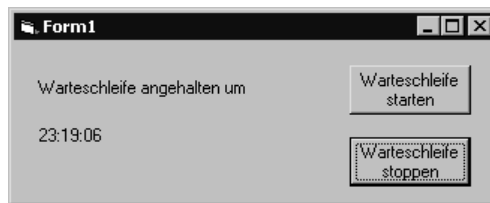


Abbildung 4.47:
Endlosschleife wurde gestoppt

5 Workshop: Prozeduren und Funktionen

Prozeduren und Funktionen können in Visual Basic in zwei grundsätzliche Kategorien eingeteilt werden.

Zum einen sind darunter die Prozeduren und Funktionen der Laufzeitbibliotheken von Visual Basic zu verstehen. Zum anderen können Sie auch eigene Prozeduren und Funktionen programmieren.

In diesem Kapitel werden zunächst einige der wichtigsten Prozeduren und Funktionen von Visual Basic vorgestellt und verwendet. Die Anzahl aller Funktionen würde den Rahmen dieses Buches sprengen. Ich möchte Ihnen empfehlen die Online-Hilfe von Visual Basic zu Rate zu ziehen, falls Sie eine Funktion benötigen, die in diesem Buch nicht beschrieben ist.

Die Online-Hilfe von Visual Basic kann auf unterschiedliche Arten verwendet werden. Wenn Sie die Online-Hilfe lesen möchten wie ein Buch, können Sie in ihr die Registerkarte Inhalt auswählen. Dort ist es möglich, die Visual Basic-Hilfe wie ein Buch zu öffnen und Kapitel für Kapitel zu lesen. Sie können diese Ansicht in Abbildung 5.1 sehen.

Prozeduren und Funktionen der Laufzeitbibliothek und eigene

Verwenden der Online-Hilfe: Registerkarte Inhalt

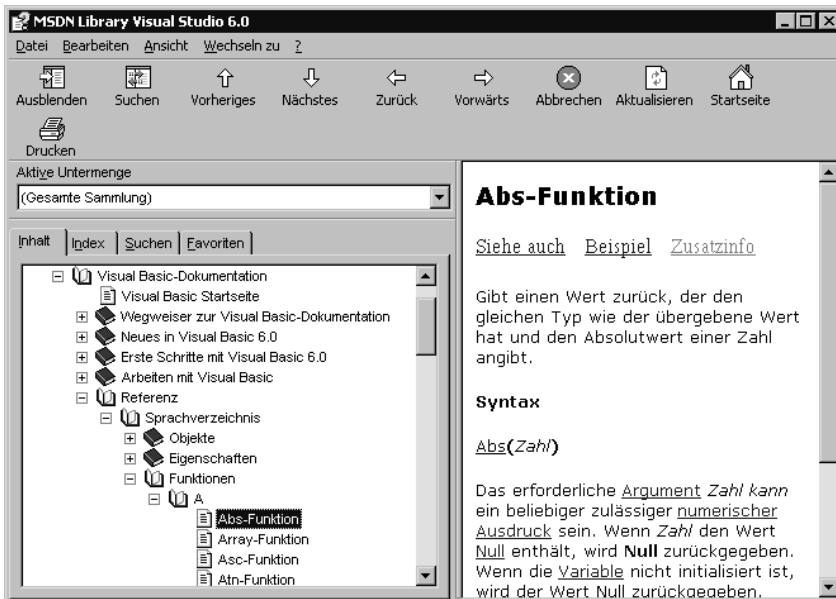


Abbildung 5.1: Ansicht: Inhalt der Visual Basic Online-Hilfe

Abbildung 5.1 zeigt die Hilfeseite der Funktion ABS, die den Absolutwert einer Zahl ermittelt. In dieser Ansicht können Sie über die Schaltflächen *Vorheriges* und *Nächstes* in der Online-Hilfe blättern wie in einem Buch.

**Registerkarte
Index**

Möchten Sie hingegen nach einer bestimmten Funktion suchen, deren Namen Sie zu kennen glauben, so können Sie die Registerkarte *Index* verwenden. Diese Registerkarte erlaubt die Eingabe eines Schlüsselwortes. Bei jedem eingegebenen Buchstaben wird die Ergebnisliste aktualisiert. Ist Ihr Suchbegriff korrekt, können Sie auf diese Weise sehr schnell eine spezielle Funktion finden.

Abbildung 5.2:
Ansicht *Index* der
Visual Basic
Online-Hilfe



In Abbildung 5.2 wurde der Suchbegriff *Sinus* eingegeben. Gleich die erste Auswahl führt zur *Visual Basic*-Funktion *Sin*, die den Sinus eines Winkels errechnet.

**Registerkarte
Suchen**

Eine dritte Möglichkeit ergibt sich schließlich durch die Registerkarte *Suchen*. Sie ermöglicht die Eingabe und logische Verknüpfung (Abbildung 5.3) von mehreren Suchbegriffen. Somit haben Sie selbst dann eine gute Möglichkeit etwas zu finden, wenn kein definierter Suchbegriff zur Verfügung steht.

In Abbildung 5.3 wurde die beiden Suchbegriffe *Dateien* und *Lesen* mit dem Operator *AND* verknüpft. Auf diese Weise werden in der Auswahlliste alle Dokumente der *Online-Hilfe* angezeigt, die beide Suchbegriffe beinhalten. Der Operator *AND* ist vor allem dann von Nutzen, wenn die Anzahl der Dokumente, die eine Suche ergibt, sehr groß ist. Die Wahrscheinlichkeit, dass viele Dokumente nicht das beinhalten, was Sie suchen, ist in diesem Fall groß. Mit *AND* und einem weiteren Suchbegriff wird die Auswahl kleiner. Meist ist es allerdings dann doch notwendig, einige der gelisteten Dokumente zu öffnen, bevor man fündig wird.

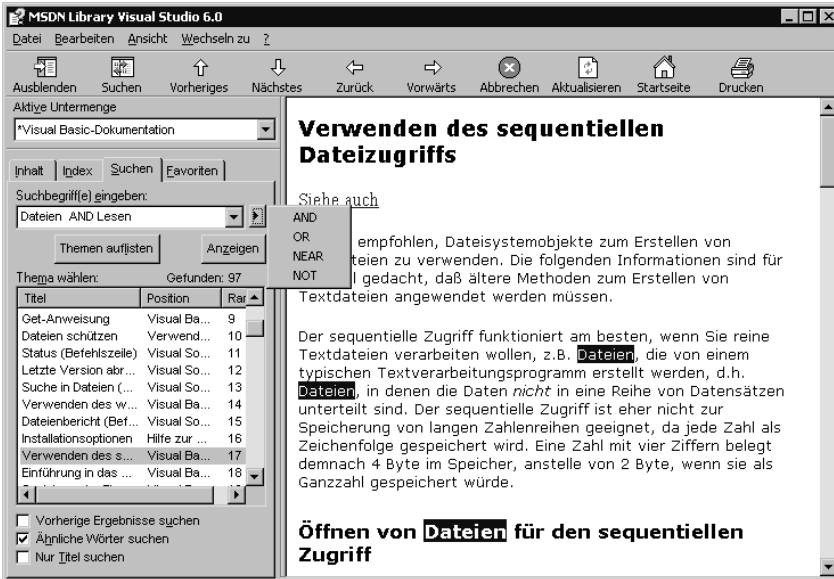


Abbildung 5.3:
Ansicht Suchen der
Visual Basic
Online-Hilfe

Im weiteren Verlauf des Kapitels werden die Mechanismen für selbst geschriebene Prozeduren und Funktionen vorgestellt. Die theoretischen Erklärungen werden am Ende durch praxisbezogene, einfache Übungen vertieft.

selbst geschriebene Prozeduren und Funktionen

In diesem Kapitel erwarten Sie Abschnitte zu den Themen:

- ▶ Verwendung des Benutzerdialogs MsgBox
- ▶ Benutzereingaben abfragen über InputBox
- ▶ Verarbeiten von Zeichenketten
- ▶ Umwandlung von Variablen und Konstanten in spezielle Datentypen
- ▶ Schreiben eigener Prozeduren und Funktionen

5.1 Der Benutzerdialog MsgBox

Zur Lösung der bisherigen Übungen wurde die Funktion MsgBox bereits in ihrer einfachsten Form verwendet, nämlich zur Ausgabe einer Benutzerinformation. Die Funktion MsgBox kann aber deutlich mehr, als wir bisher genutzt haben. So ist es unter anderem möglich, das Aussehen des Dialogs und seine Funktion durch Verwendung von weiteren Übergabeparametern zu verändern.

[Ergebniswert =] MsgBox(Meldung [, Schaltflächen] [, Titel])
[Hilfdatei, Hilfekontext])

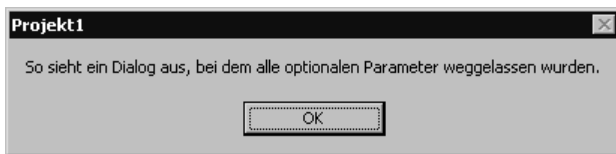
Syntax

Bei der Funktion `MsgBox` sind alle Parameter und der Rückgabewert der Funktion optional, d.h. sie können auch weggelassen werden. Diese Form wurde bisher benutzt. Der Aufruf sieht dann folgendermaßen aus:

```
MsgBox "So sieht ein Dialog aus, bei dem alle optionalen Parameter weggelassen wurden."
```

Die Ausgabe dieser Programmzeile sehen Sie in Abbildung 5.4. Es ist ein Dialog mit einem Standardtitel, dem übergebenen Text als Info und einer OK-Schaltfläche. Wird diese betätigt, wird der Dialog geschlossen und das Programm läuft mit der folgenden Programmzeile weiter.

Abbildung 5.4:
Standard-Dialog
mit Default-
Parametern der
`MsgBox`-Funktion



Die maximale Länge des Parameters *Meldung* ist auf 1024 Zeichen begrenzt. Sie kann bei Verwendung anderer Schriftarten eventuell etwas kürzer sein. Längere Meldungen werden ab dieser Stelle abgeschnitten.

Die `MsgBox`-Funktion bricht einen längeren Text automatisch um und stellt diesen mehrzeilig dar. Wenn Sie die Formatierung beeinflussen möchten, sollten Sie die vordefinierten Konstanten `vbCrLf` und `vbTab` in die auszugebende Zeichenkette einbauen.

Parameter Schaltflächen

Der Parameter *Schaltflächen* beinhaltet im Wesentlichen die Möglichkeiten der Funktion `MsgBox`. Der Parameter ist im Grunde ein einfacher numerischer Wert. Die Funktion `MsgBox` wertet allerdings die einzelnen Bits dieser Zahl aus, wobei jedes Bit eine eigene Bedeutung hat. Aber keine Angst, Sie müssen nicht mit Bits und Binärzahlen rechnen, um den Parameter Schaltflächen zu setzen.

Für diesen Zweck sind in Visual Basic Konstanten vordefiniert, die einfach mit dem Operator `+` verknüpft werden. Als Resultat der Addition sind im Ergebnis die Bits beider Ausgangswerte gesetzt. Sie haben also mit den Bitoperationen an sich gar nichts zu tun.

Die Konstanten für den Parameter Schaltflächen lassen sich in vier Gruppen einteilen:

- ▶ Konstanten, die Anzahl und Art der verwendeten Schaltflächen festlegen.
- ▶ Konstanten, die das verwendete Grafiksymbold festlegen.
- ▶ Konstanten, die eine der gezeigten Schaltfläche zur Befehlsschaltfläche machen.

- Konstanten, die für spezielle Einstellungen, wie beispielsweise die Modalität des Dialog, verwendet werden.

In den folgenden Abschnitten werden die definierten Konstanten der einzelnen Gruppen vorgestellt.

Anzahl und Art der verwendeten Schaltflächen

Tabelle 5.1 beinhaltet alle Konstanten, die verwendet werden können um die Anzahl und Art der im Dialogfeld angezeigten Schaltflächen zu manipulieren.

Konstante	Wert	Bedeutung
VbOKOnly	0	Nur OK-Button anzeigen (Voreinstellung)
VbOKCancel	1	OK und Abbrechen anzeigen
VbAbortRetryIgnore	2	Beenden, Wiederholen und Ignorieren anzeigen
VbYesNoCancel	3	Ja, Nein und Abbrechen anzeigen
VbYesNo	4	Ja und Nein anzeigen
VbRetryCancel	5	Wiederholen und Abbrechen anzeigen

Tabelle 5.1:
Schaltflächen in
Message-Boxen

Die folgende Programmzeile zeigt die Anwendung einer dieser Konstanten um einen Dialog anzuzeigen, der eine Ja/Nein-Entscheidung zulässt.

MsgBox "Möchten Sie einen Kaffee trinken?", vbYesNo

Den Dialog, der aus dieser Programmzeile resultiert, sehen Sie in Abbildung 5.5.

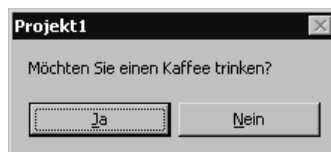


Abbildung 5.5:
Dialog mit mehreren
Schaltflächen

Das verwendete Grafiksymbol

Durch Verwendung der Konstanten aus Tabelle 5.2 kann ein MsgBox-Dialog mit einer Grafik kombiniert werden. Diese ist allerdings nicht frei definierbar, sondern kann nur aus einer Liste von vier vordefinierten Grafiken gewählt werden.

Im Parameter Schaltfläche darf nur eine der folgenden Konstanten verwendet werden, da die Funktion MsgBox nicht in der Lage ist mehr als eine Grafik anzuzeigen.

Tabelle 5.2:
Grafiksymbole in
Message-Boxen

Konstante	Wert	Bedeutung
VbCritical	16	Stop-Symbol anzeigen (schwere Fehler)
VbQuestion	32	Fragezeichen-Symbol anzeigen (Rückfragen)
VbExclamation	48	Ausrufezeichen-Symbol anzeigen (Warnungen)
VbInformation	64	Informations-Symbol anzeigen (Informationen)

Die vier definierten Konstanten sollten zweckentsprechend verwendet werden. D.h. wenn der Anwender nur informiert werden soll, so sollte die Konstante *vbInformation* verwendet werden, ist ein schwerer Fehler aufgetreten, so sollte die Konstante *vbCritical* zum Einsatz kommen, usw.

Auf diese Weise hat der Anwender bereits durch die Grafik einen Hinweis, ob er sich Sorgen machen sollte oder ob es sich wieder einmal nur um eine dieser wertvollen Sicherheitsabfragen handelt (»Sind Sie sich auch wirklich ganz, ganz, ganz sicher, dass Sie das Programm beenden möchten?«).

Die folgende Programmzeile demonstriert die Anwendung dieser Konstanten für einen wirklich kritischen Zustand:

```
MsgBox "Die Kaffeemaschine ist defekt !", vbCritical
```

Die Ausgabe dieses Dialogs ist in Abbildung 5.6 zu sehen.

Abbildung 5.6:
MessageBox mit
drei Schaltflächen
und einem Symbol
aufrufen



Festlegung der Standard-Befehlsschaltfläche

Standard-Befehlsschaltfläche ist die Bezeichnung der Schaltfläche, die direkt nach dem Start des Dialogs aktiviert ist. Wird keine der Konstanten aus Tabelle 5.3 verwendet, so ist die erste Schaltfläche des Dialogs nach dem Start aktiviert, d.h. sie hat den Fokus und wird bei Betätigen der Taste `ENTER` ausgeführt.

unkritische Schaltflächen aktivieren

Wenn Sie verhindern möchten, dass der Anwender versehentlich durch Drücken der Taste ENTER eine Funktion ausführt, die kritisch ist, oder einen Programmzustand unwiderruflich ändern, sollten Sie die Standard-Befehlsschaltfläche auf eine unkritische Auswahl stellen.

Da die Funktion *MsgBox* maximal vier Schaltflächen darstellen kann, gibt es auch genau vier mögliche Einstellungen. Auch bei dieser Gruppe von Konstanten darf aus verständlichen Gründen nur eine im Parameter Schaltfläche verwendet werden.

Konstante	Wert	Bedeutung
VbDefaultButton1	0	Erste Befehlsschaltfläche ist Voreinstellung
VbDefaultButton2	256	Zweite Befehlsschaltfläche ist Voreinstellung
VbDefaultButton3	512	Dritte Befehlsschaltfläche ist Voreinstellung
VbDefaultButton4	768	Vierte Befehlsschaltfläche ist Voreinstellung

Tabelle 5.3:
Standardschaltflächen in Message-Boxen

Die folgende Programmzeile setzt die dritte Schaltfläche als Standard-Schaltfläche:

```
MsgBox "Der Betrag übersteigt Ihr Limit!", vbAbortRetryIgnore + vbInformation + vbDefaultButton3
```

In dieser Zeile wurden mehrere bereits besprochene Konstanten durch den Operator + verknüpft. Es sollen eine Grafik und drei Schaltflächen angezeigt werden. Die dritte Schaltfläche wird als Standard-Schaltfläche definiert. Das Ergebnis dieses Dialogs sehen Sie in Abbildung 5.7.

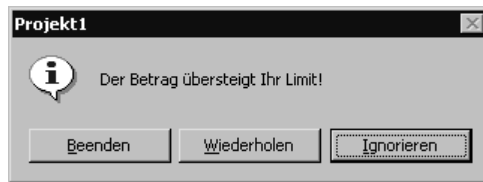


Abbildung 5.7:
Standard-Schaltfläche des Dialogs festlegen

Spezielle Einstellungen

Mit Hilfe der Werte der vierten und letzten Gruppe können einige Spezialeinstellungen festgelegt werden. Die hierzu notwendigen Konstanten sind in Tabelle 5.4 zu finden.

Konstante	Wert	Bedeutung
VbApplicationModal	0	Macht die Message-Box anwendungsmodal; d.h. der Anwender muss auf sie reagieren, bevor er die Arbeit mit der jeweiligen Anwendung fortsetzen kann.
VbSystemModal	4096	Macht die Message-Box systemmodal; wodurch alle Anwendungen unterbrochen werden, bis der Anwender auf die Nachricht reagiert.
VbMsgBoxHelpButton	16384	Fügt der Message-Box eine Hilfeschaltfläche hinzu.
VbMsgBoxSetForeground	65536	Legt das Message-Box-Fenster als Vordergrundfenster fest.

Tabelle 5.4:
Modalität von Message-Boxen

Konstante	Wert	Bedeutung
VbMsgBoxRight	524288	Richtet Text rechtsbündig aus.
VbMsgBoxRtlReading	1048576	Legt fest, dass der anzuzeigende Text von rechts nach links angezeigt wird.

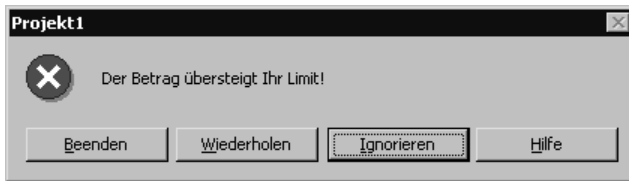
Von den Konstanten dieser Gruppe können mehrere gleichzeitig im Parameter Schaltfläche verwendet werden. Lediglich die Konstanten *vbApplicationModal* und *vbSystemModal* schließen sich gegenseitig aus.

Mit der folgenden Programmzeile wird ein Dialog mit einer Hilfe-Schaltfläche dargestellt:

```
MsgBox " Der Betrag übersteigt Ihr Limit!", vbAbortRetryIgnore +  
vbCritical + vbDefaultButton3 + vbMsgBoxHelpButton
```

Das Ergebnis dieser Anweisung ist in Abbildung 5.8 zu sehen.

Abbildung 5.8:
Eine Schaltfläche
für eine
Online-Hilfe



Wenn Sie die Hilfeschaltfläche anklicken werden Sie feststellen, dass nichts passiert. Um tatsächlich eine Hilfeseite zu öffnen, müssen weitere Parameter der Funktion *MsgBox* erklärt werden.

Die weiteren Übergabeparameter der MsgBox-Funktion

Im Parameter *Titel* wird der Text der Titelzeile des Dialogs festgelegt. Wird kein expliziter Titel angegeben, so zeigt Visual Basic den jeweiligen Projektnamen als Standardtitel an.



Sie können auch einen *Titel* für eine *MsgBox*-Funktion übergeben, die keine speziellen Werte für den Parameter *Schaltfläche* bekommt. In diesem Fall wird der Platz in der Übergabeliste für den Parameter *Schaltflächen* leer gelassen, d.h. es stehen zwei Kommas direkt hintereinander. Eine *MsgBox*-Anweisung, die diese Funktion nutzt, sieht folgendermaßen aus:

```
MsgBox "Ein guter Titel ziert alles.", , "Workshop"
```

Den Dialog, der durch obigen Aufruf erzeugt wird, sehen Sie in Abbildung 5.9.

Das Auslassen optionaler Parameter in einer Parameterliste wird sehr häufig benötigt und ist nicht der Funktion *MsgBox* vorbehalten, sondern bei allen Visual Basic-Funktionen die übliche Vorgehensweise.



Abbildung 5.9:
Optionale Parameter in Übergabeparameterliste überspringen

Mit dem Parameter *Hilfdatei* kann für den Dialog der Name einer Hilfedatei definiert werden. Diese kann dann für die Anzeige einer kontextsensitiven Hilfe genutzt werden, wenn die Schaltfläche Hilfe angezeigt und ausgewählt wurde.

Der Parameter *Hilfekontext* ermöglicht die Angabe einer Kontextnummer des jeweiligen Hilfethemas. Die Kontextnummer sorgt dafür, dass aus einer großen Hilfedatei die zum Dialog passende Hilfeseite automatisch geladen wird.

Die Rückgabewerte der MsgBox-Funktion

Sobald mehr als eine Schaltfläche im Dialog angezeigt wird, ist eine Rückmeldung der *MsgBox*-Funktion abfragbar. Somit hat der Programmierer die Möglichkeit, die Anwenderauswahl festzustellen und dann im Programm entsprechend darauf zu reagieren.

Der Rückgabewert einer *MsgBox*-Funktion kennzeichnet die Schaltfläche, welche vom Anwender betätigt wurde.

Das Ergebnis ist ein numerischer Wert des Datentyps *Integer*. Sie müssen diese numerischen Werte allerdings nicht direkt benutzen, sondern können auf vordefinierte Konstanten zurückgreifen, die Ihr Programm leichter lesbar machen.

Die möglichen Rückgabewerte und deren zugehörigen Konstanten sind in Tabelle 5.5 zu sehen.

Konstante	Rückgabewert	Ausgewählte Schaltfläche
VbOK	1	OK
VbCancel	2	Abbruch
VbAbort	3	Abbruch
VbRetry	4	Wiederholen
VbIgnore	5	Ignorieren
VbYes	6	Ja
VbNo	7	Nein

Tabelle 5.5:
Rückgabewerte von Message-Boxen

Folgender Programmcode zeigt die Zuweisung und Auswertung der Rückgabe einer *MsgBox*-Funktion:

```
Dim Ergebnis As Integer
Ergebnis = MsgBox("Der Betrag übersteigt Ihr Limit!",
```

```
vbAbortRetryIgnore + vbInformation + vbDefaultButton1)
Select Case Ergebnis
    Case vbAbort ' Beenden
        MsgBox "Sehr vernünftig."
    Case vbRetry ' Wiederholen
        MsgBox "Na gut."
    Case vbIgnore ' Ignorieren
        MsgBox "Von mir aus."
End Select
```



Sobald die *MsgBox*-Funktion in einer Zuweisung verwendet wird, muss die Übergabeparameterliste in Klammern gesetzt werden.

Mit einer *Select Case*-Struktur kann der Rückgabewert am einfachsten ausgewertet werden.

Der Dialog wird im Beispiel mit einer Grafik gezeigt, die ihn als informative Meldung kennzeichnet. Abbildung 5.10 zeigt den Dialog.

Abbildung 5.10:
MsgBox-Dialog als
informativer
Hinweis



Wird die Schaltfläche *Ignorieren* betätigt, so erscheint gemäß der Auswertung in der *Select Case*-Struktur der Dialog aus Abbildung 5.11.

Abbildung 5.11:
Es wurde die
Schaltfläche *Ignorieren*
betätigt.



5.1.1 Übung: Formatierte Ausgabe mit der Funktion *MsgBox*

Schreiben Sie ein Programm, welches in einem *MsgBox*-Dialog zweizeilig folgenden Text ausgibt:

Bier 2 Gläser

Kaffee 3 Tassen

Die Ausgabe muss dabei so formatiert werden, dass die Zahlen in einer Spalte untereinander stehen.

Lösung

Um diese Übung zu lösen, müssen die vordefinierten Konstanten `vbCrLf` und `vbTab` verwendet werden.

Die zweizeilige Ausgabe des Textes wird durch `vbCrLf` erzwungen. Diese wird hinter dem Wort *Gläser* eingefügt.

Da die Wörter Bier und Kaffee nicht die gleiche Länge haben, muss in diesem Fall die Konstante `vbTab` verwendet werden. Diese Konstante sorgt dafür, dass das folgende Zeichen an der nächsten Tab-Position beginnt.

Der Versuch, das Gleiche durch die Eingabe von zusätzlichen Leerzeichen zu erreichen, ist normalerweise nicht sinnvoll, da für die Schriftart des Dialogs eine proportionale Schrift verwendet wird, d.h. unterschiedliche Zeichen des Satzes haben unterschiedliche Länge.

Folgende Programmzeile zeigt daher auch kein befriedigendes Ergebnis, wie in Abbildung 5.12 zu sehen ist.

```
MsgBox "Bier  2 Gläser" & vbCrLf & "Kaffee 3 Tassen"
```



**proportionale
Schrift macht
Ausrichtung ohne
Tabulator
schwierig**

Abbildung 5.12:
Auffüllen mit Leer-
zeichen funk-
tioniert nicht

Die folgende Programmzeile zeigt stattdessen die Verwendung der Konstanten `vbTab`:

```
MsgBox "Bier" & vbTab & "2 Gläser" & vbCrLf & "Kaffee" & vbTab & "3  
Tassen"
```

In Abbildung 5.13 können Sie sehen, dass hiermit ein tadelloses Ergebnis zustande kommt.



Abbildung 5.13:
Eine tadellos for-
mattierte Ausgabe
mit MsgBox

5.1.2 Übung: Dialog mit Rückgabe und Auswertung

In einem Programm soll eine Datei gelöscht werden. Vor dem Löschen der Datei soll eine Abfrage als Warnhinweis angezeigt werden, dass die Datei nicht wiederhergestellt werden kann. Der Anwender soll die Möglichkeit haben das Löschen der Datei abzubrechen.

Für jede Entscheidung wird eine informative Meldung ausgegeben.



Lösung

Die Lösung dieser Übung entspricht im Wesentlichen dem Beispiel des Abschnitts *Die Rückgabewerte der MsgBox-Funktion*. Grundsätzlich ist eine Variable des Datentyps Integer notwendig, die den Rückgabewert der Funktion *MsgBox* speichert. Zudem wird eine *Select...Case*-Struktur benötigt, die für die Auswertung des Rückgabewertes verwendet wird.

Ansonsten handelt es sich bei der Übung im Wesentlichen um die Gestaltung des Textes, eines Titels und der Werte für den Parameter *Schaltflächen*.

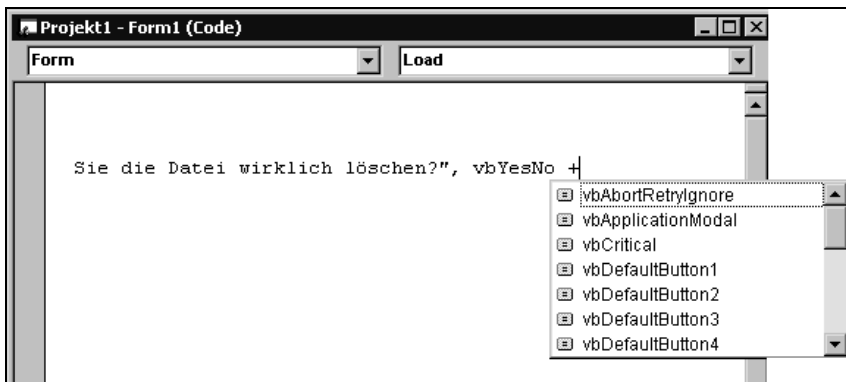


intelligente Eingabehilfe

Bei der Programmierung der Aufgabe haben Sie sicher festgestellt, dass Ihnen Visual Basic bei der Eingabe des Parameters *Schaltflächen* tatkräftig zur Seite steht. Abbildung 5.14 zeigt die kontextsensitive Eingabehilfe des Visual Basic-Programmeditors. Da die Namen der Konstanten aussagekräftig sind, ist ein Nachschauen in der Online-Hilfe oft unnötig.

Beachten Sie bitte, dass Visual Basic nicht nur nach dem Schreiben des Kommas die Eingabehilfe aufblendet, sondern auch nach dem Schreiben eines *Plus*-Zeichens, so dass alle Konstanten des Parameters über die Eingabehilfe ausgewählt werden können.

Abbildung 5.14:
Die vordefinierten
Konstanten in der
Entwicklungs-
umgebung



In der Übung wird verlangt den Anwender zu warnen, dass eine Datei gelöscht werden soll. Er soll die Möglichkeit haben diesen Vorgang abzubrechen, d.h. er entscheidet in diesem Dialog, ob die Datei gelöscht werden soll oder nicht. Der Dialog kann also bei entsprechender Fragestellung mit den Schaltflächen Ja und Nein ausgestattet werden. Die notwendige Konstante hierfür ist *vbYesNo*.

Warnmeldung

Als zweite Bedingung soll dieser Hinweis eine Warnmeldung darstellen. Daher wird eine Grafik angezeigt, die den Dialog zusätzlich zum Text als Warnmeldung identifiziert. Die notwendige Konstante hierfür lautet *vbExclamation*.

Die Frage, die gestellt wird, lautet: Möchten Sie die Datei wirklich löschen? Die ungefährlichere Antwort ist daher sicherlich *Nein*. Per Default wird allerdings

die erste Schaltfläche, also *Ja*, als Standard-Befehlsschaltfläche verwendet. Um dies zu ändern, wird die zusätzliche Konstante *vbDefaultButton2* hinzugefügt. Diese macht die Schaltfläche 2, also *Nein*, zur Standard-Befehlsschaltfläche.

Die folgenden Programmzeilen implementieren den ersten Teil der Übung:

```
Private Sub Form_Load()
Dim Ergebnis As Integer
Ergebnis = MsgBox("Wenn die Datei xyz gelöscht wird, kann sie nicht wieder hergestellt werden!" & vbCrLf & vbCrLf & "Möchten Sie die Datei wirklich löschen?", vbYesNo + vbExclamation + vbDefaultButton2, "Warnung")
```

Wird diese Anweisung ausgeführt, erscheint der Dialog aus Abbildung 5.15.

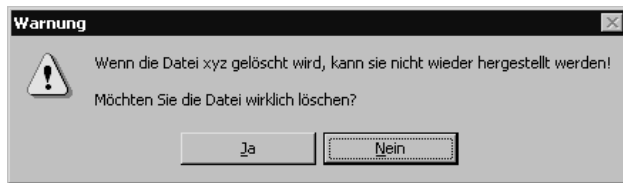


Abbildung 5.15:
Eine Warnmeldung
mit Ja/Nein-
Entscheidung

Beachten Sie bitte, dass in Abbildung 5.15 der informative Text und die eigentliche Frage durch eine Leerzeile getrennt sind. Dies erreichen Sie durch das Einfügen von zwei Konstanten *vbCrLf*. Die erste schließt den informativen Text ab, die zweite generiert eine Leerzeile.



Wenn Sie eine Entscheidung von einem Anwender verlangen und vor der Frage ein erklärender Text notwendig ist, sollten Sie immer eine Leerzeile zwischen Text und Frage einschieben. So hat der Anwender die Frage, die beantwortet werden soll, besser im Blick.

In der Fortsetzung des Programmcodes muss jetzt der Rückgabewert von *MsgBox* ausgewertet werden. Als Ergebnis kommt auf Grund des Dialogs entweder die Konstante *vbYes* oder die Konstante *vbNo* in Frage.

Die notwendige *Select...Case*-Struktur hat also lediglich zwei Fälle zu unterscheiden, wie Sie aus folgenden Programmzeilen ersehen können:

```
Select Case Ergebnis
Case vbYes
MsgBox "Datei wurde gelöscht.", vbInformation
Case vbNo
MsgBox "Datei wurde nicht gelöscht.", vbInformation
End Select
End Sub
```

Falls der Anwender nach dem Start des Dialog die Taste **ENTER** drückt, so wird die Standard-Befehlsschaltfläche, also *Nein*, aktiviert und der Dialog aus Abbildung 5.16 ausgegeben.

Auswertung der Rückgabe

Standard-Befehlsschaltfläche ist unkritisch

Abbildung 5.16:
Eine Information:
Die Datei wurde
nicht gelöscht



Dieser Dialog wurde als Information gekennzeichnet, durch die Verwendung der Konstanten *vbInformation*.

Falls der Anwender sich dazu entscheidet, die Datei zu löschen, so muss er die Schaltfläche *Ja* im Eingangsdialog auswählen. Der dann eingeblendete Dialog ist ebenfalls eine rein informative Meldung. Sie sehen ihn in Abbildung 5.17.

Abbildung 5.17:
Die Datei wurde
gelöscht



5.2 Der Benutzerdialog InputBox

Die *MsgBox*-Funktion ist ein Dialog um Informationen zu visualisieren und einfache Entscheidungen abzufragen. Wenn der Anwender jedoch einen Wert eingeben soll, beispielsweise nach seinem Namen gefragt wird, so ist dies mit der Funktion *MsgBox* nicht möglich.

Genau für diesen Fall gibt es die Funktion *InputBox*. Sie erlaubt die Eingabe von alphanummerischen Daten. Diese können dann, je nach Verwendungszweck im Programm, in beliebige andere Datentypen konvertiert werden.

Syntax `Ergebnis = InputBox(Meldung [, Titel] [, Vorgabewert] [, x] [, y] [, Hilfedatei, Hilfekontext]`

Der Parameter *Meldung* wird als erklärender Text im *InputBox*-Dialog dargestellt. Der Benutzer wird mit diesem Text über die Art und den Zweck seiner Eingabe aufgeklärt. Dies ist vor allem deshalb wichtig, weil der Anwender beliebige Zeichenketten eingeben kann. Wird jedoch ein Betrag gefordert, muss der Anwender informiert werden, dass er eine Zahl eingeben soll.

Der Parameter *Titel* legt den Inhalt der Titelzeile des Dialogs fest. Er ist optional. Wird er nicht festgelegt, so verwendet Windows den Projekt- bzw. Programmnamen als Standardwert.

Mit dem Parameter *Vorgabewert* wird der Inhalt der Eingabe-TextBox nach dem Start des Dialogs festgelegt. Falls der Anwender keine Eingaben macht, sondern den Dialog einfach nur bestätigt, wird dieser Wert auch wieder zurückgegeben. Er entspricht also dem Standardwert für die Eingabe.

Der Parameter *Vorgabewert* ist optional. Wird er nicht verwendet, ist die Eingabe-Textbox nach dem Start des Dialogs leer.

Verwenden Sie wann immer möglich einen Vorgabewert. Zusätzlich zu seiner Eigenschaft als Standard-Rückgabe stellt er eine zusätzliche Eingabehilfe für den Anwender dar. Denn wenn der Vorgabewert numerisch ist, wird der Anwender dies bemerken und auch einen numerischen Wert eingeben.



Die Parameter *x* und *y* bestimmen die Position des Dialogs in Twips bezogen auf die linke obere Bildschirmcke. Diese Parameter sind ebenfalls optional. Werden sie weggelassen, so wird der Dialog horizontal auf dem Bildschirm zentriert. Vertikal wird der Dialog als Standard etwa ein Drittel unterhalb der oberen Bildschirmkante angezeigt.

Die Parameter *Hilfdatei* und *Hilfekontext* haben die gleiche Bedeutung wie bei der *MsgBox*-Funktion.

Rückgabe des Ergebnisses

Der Rückgabewert eines *InputBox*-Dialogs ist in jedem Fall eine Variable des Datentyps *String*.

Falls der Dialog mit der Schaltfläche *OK* beendet wurde, wird in diesem String der Inhalt der Eingabe-Textbox zurückgeliefert.

Wird der Dialog mit der Schaltfläche *Abbrechen* beendet, so ist das Ergebnis bzw. der Rückgabewert ein Leerstring.

Folgendes Beispiel zeigt die Funktionsweise des *InputBox*-Dialogs:

```
Private Sub Form_Load()  
Dim ergebnis As String  
ergebnis = InputBox("Bitte geben Sie Ihren Namen ein.")  
MsgBox "Ihr Name ist " & ergebnis  
End Sub
```



Beim Aufruf der *InputBox*-Funktion dieses Programmbeispiels wird der Dialog aus Abbildung 5.18 angezeigt.

Beachten Sie bitte die Position, die sich ergibt, wenn keine Position angegeben wurde. Er befindet sich horizontal mittig, aber vertikal etwas nach oben verschoben. Hier handelt es sich um die Angabe *ein Drittel vom oberen Bildschirmrand entfernt*.

Inputbox ist standardmäßig nicht vertikal mittig

In Abbildung 5.18 wurde bereits die Eingabe *Heinz* gemacht. Wird jetzt die Schaltfläche *Abbrechen* des Dialogs betätigt, erhalten Sie als Ausgabe der folgenden *MsgBox*-Funktion den Dialog aus Abbildung 5.19.

Obwohl eine Eingabe gemacht wurde, lieferte die *InputBox*-Funktion einen Leerstring zurück. Unser Beispiel ist auf diese Möglichkeit nicht vorbereitet und gibt daher den sinnlosen Dialog aus Abbildung 5.19 aus.

Abbildung 5.18:
Der Inputbox-
Dialog und seine
Standardposition
auf dem Bildschirm

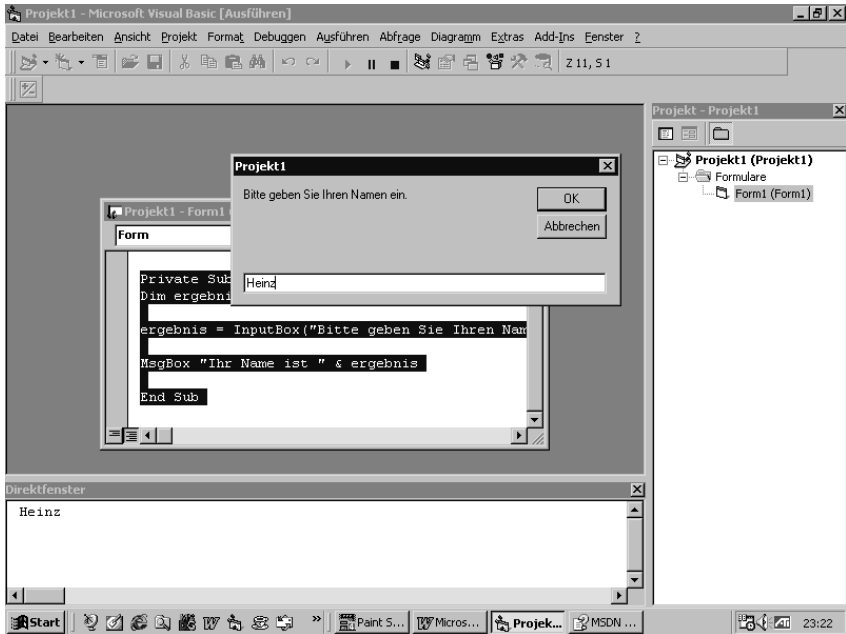


Abbildung 5.19:
Es wurde die
Schaltfläche
Abbrechen
gedrückt



Wenn Sie die InputBox-Funktion für Benutzereingaben verwenden, muss Ihr Programm auf den Abbruch des Dialogs und/oder falsche Eingaben vorbereitet sein.

Wird in obigem Programm die Schaltfläche OK des InputBox-Dialogs gedrückt, so erscheint im folgenden MsgBox-Dialog auch der eingegebene Name. Sie können dies in Abbildung 5.20 vergleichen.

Abbildung 5.20:
Es wurde die
Schaltfläche OK
gedrückt



5.2.1 Übung: Verwenden der InputBox-Funktion



Schreiben Sie ein Programm, welches zwei Zahlen miteinander multipliziert und das Ergebnis in einem Dialog ausgibt. Die Operanden werden vom Benutzer abgefragt.

Positionieren Sie den ersten Eingabedialog am oberen, linken Bildschirmrand. Verwenden Sie bei diesem Eingabedialog keinen Titel.

Positionieren Sie den zweiten Eingabedialog horizontal mittig und vertikal etwa ein Drittel unterhalb der oberen Bildschirmkante.

Lösung

Für die Lösung dieser Übung werden zwei Eingabedialoge und ein Ausgabedialog benötigt. Die Rückmeldungen der Eingabedialoge werden für die Berechnung in Zahlen umgewandelt. Das Ergebnis der Berechnung könnte in einer eigenen Variablen gespeichert werden. Da es im Programm allerdings nur für die Ausgabe in einem Dialog benötigt wird, kann die Berechnung auch direkt im Ausgabedialog-Aufruf ausgeführt werden. Es werden also insgesamt nur drei Variablen benötigt.

```
Private Sub Form_Load()  
Dim operand1 As Double  
Dim operand2 As Double  
Dim rueckgabe As String
```

Das Einlesen und Umwandeln des ersten Operanden erfolgt mit folgenden Programmzeilen:

```
rueckgabe = InputBox("Geben Sie bitte den ersten Operanden der  
Multiplikation ein.", , "1", 0, 0)  
operand1 = Cdbl(rueckgabe)
```

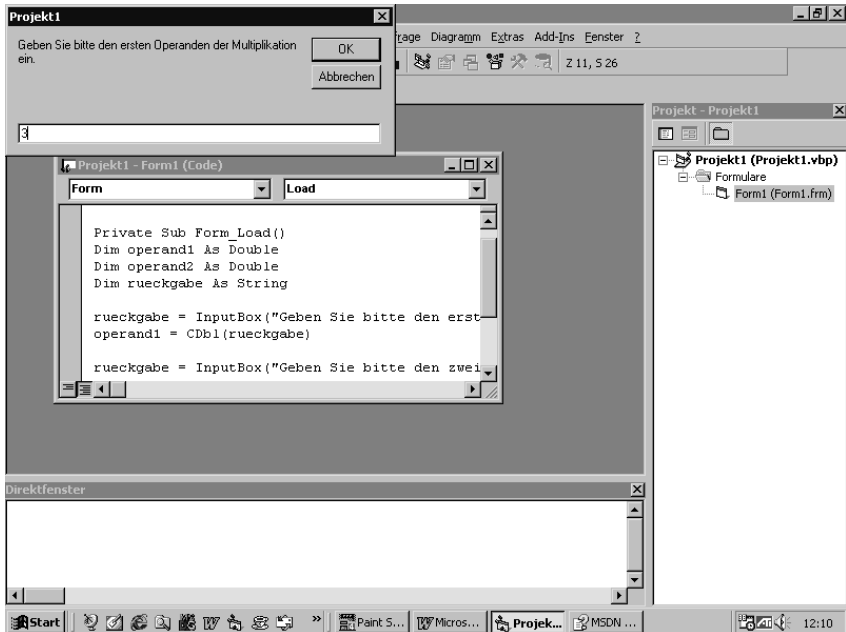
Als Position (Parameter x und y) wurde die Bildschirmecke links oben gewählt. Da die Positionsangaben relativ zu diesem Ursprung gerechnet werden, ist diese Position durch die einfache Angabe des Werts 0 für beide Parameter zu erreichen. In Abbildung 5.21 können Sie sehen, dass die Positionierung wie gewünscht funktioniert.

Wie in der Übungsbeschreibung gefordert wurde kein Titel angegeben. Folgerichtig erscheint in der Titelzeile der Programm- bzw. Projektname. Da der Titel vor den Parametern für die Position den Dialogs in der Parameterliste steht, muss er ausgelassen werden. Dies wird erreicht, indem zwischen den beiden Kommas, die den Titel umgeben, kein Wert eingetragen wird.

Obwohl es die Übung nicht verlangt und kein Wert existiert, welcher öfter als andere für die Berechnung verwendet werden könnte, wurde ein Default gesetzt. Der einzige Zweck dieses Defaultwertes ist, dem Anwender zu zeigen, dass numerische Daten eingegeben werden sollen.



Abbildung 5.21:
Eingabedialog
wurde positioniert



Um den zweiten Eingabedialog auf den Bildschirm zu bringen und die Rückgabe zu konvertieren, werden folgende Programmzeilen ausgeführt:

```
rueckgabe = InputBox("Geben Sie bitte den zweiten Operanden der
Multiplikation ein.", "Multiplikation", "1")
operand2 = CDb1(rueckgabe)
```

Die Ausgabe dieses Eingabedialogs sehen Sie in Abbildung 5.22.

Beim zweiten Eingabedialog wurde sowohl ein Titel als auch ein Defaultwert gesetzt. Aber wo sind die Parameter x und y? Die Lösung ist sehr einfach. Die von der Übung geforderte Position ist die Standard-Position der *InputBox*-Funktion. Aus diesem Grund werden die Parameter x und y einfach weggelassen.

Zu guter Letzt müssen die eingelesenen Werte nur noch multipliziert und ausgegeben werden. Dies erfolgt durch folgende Programmzeile:

```
MsgBox "Das Ergebnis der Berechnung " & operand1 & " * " & operand2 &
" ist:" & vbCrLf & operand1 * operand2
```

automatische Konvertierung

Die Berechnung wird in diesem Beispiel innerhalb des Aufrufs der Funktion *MsgBox* durchgeführt. Das Ergebnis ist ebenfalls eine Variable des Datentyps *Double*. Da aus dem Kontext (Verkettung von Zeichenketten) jedoch klar ist, dass dieses Ergebnis als String verwendet werden muss, führt Visual Basic automatisch eine Konvertierung durch.

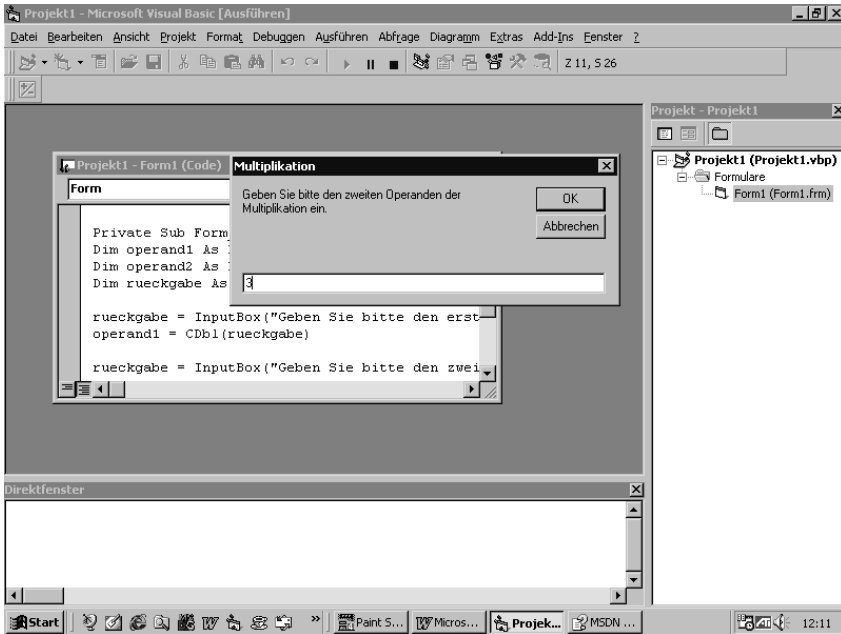


Abbildung 5.22:
Position des zweiten Eingabedialogs

Das Resultat dieser Programmzeile sehen Sie in Abbildung 5.23.

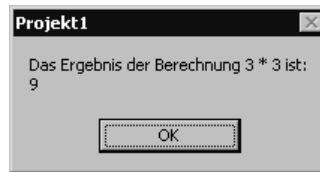


Abbildung 5.23:
Berechnung des Ergebnisses und automatische Konvertierung

Ist das Programm jetzt wirklich fertig?

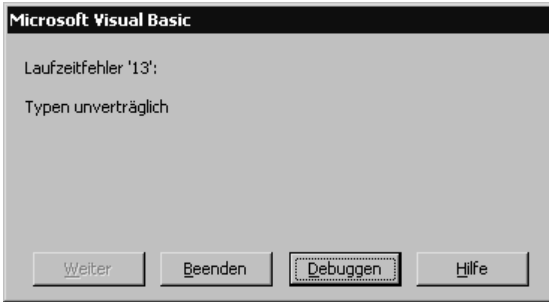
Die *InputBox*-Funktion ermöglicht dem Anwender die Eingabe beliebiger Zeichen. Es wird bei der Eingabe keinerlei Typprüfung unternommen. Das bedeutet, selbst wenn Sie den Anwender auffordern einen numerischen Wert einzugeben, können Sie nicht sicher sein, ob er das auch wirklich macht.

Gibt der Anwender in unserem Programm aber beispielsweise eine Zeichenkette ein, erhalten Sie sofort einen Laufzeitfehler, wie in Abbildung 5.24 zu sehen ist.

Wird die *InputBox*-Funktion zur Eingabe verwendet, kann der Anwender beliebige Zeichenketten eingeben. Daher muss in jedem Fall die Rückgabe überprüft oder eine Fehlerbehandlungsroutine aktiviert werden.



Abbildung 5.24:
Falsche Eingaben
führen zu einem
Laufzeitfehler



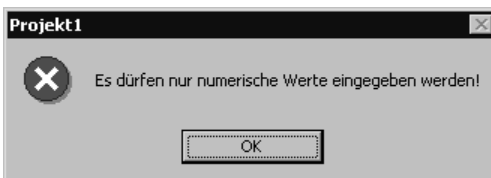
Fehlerbehandlungsroutine programmieren

Um das Programm zu vervollständigen und absturzsicher zu machen, wird der Programmcode also noch um eine Fehlerbehandlungsroutine erweitert. Die geänderte Prozedur sehen Sie in folgendem Programmauszug:

```
Private Sub Form_Load()  
Dim operand1 As Double  
Dim operand2 As Double  
Dim rueckgabe As String  
On Error GoTo eingabe_err  
rueckgabe = InputBox("Geben Sie bitte den ersten Operanden der  
Multiplikation ein.", , "1", 0, 0)  
operand1 = CDb1(rueckgabe)  
rueckgabe = InputBox("Geben Sie bitte den zweiten Operanden der  
Multiplikation ein.", "Multiplikation", "1")  
operand2 = CDb1(rueckgabe)  
MsgBox "Das Ergebnis der Berechnung " & operand1 & " * " & operand2 &  
" ist:" & vbCrLf & operand1 * operand2  
Exit Sub  
eingabe_err:  
    MsgBox "Es dürfen nur numerische Werte eingegeben werden!",  
vbCritical  
End Sub
```

Wenn der Anwender jetzt eine falsche Eingabe macht, wird das Programm nicht durch einen Laufzeitfehler beendet, sondern nach der Ausgabe der Fehlermeldung aus Abbildung 5.25 kann das Programm normal fortgesetzt werden.

Abbildung 5.25:
Der Eingabefehler
wurde abgefangen



5.3 Funktionen zur Verarbeitung von Zeichenketten

Die Verarbeitung von Zeichenketten ist eine sehr wichtige, weil häufig auftretende Programmierarbeit. Im diesem Kapitel werden die Funktionen vorgestellt, die Visual Basic zu diesem Thema zur Verfügung stellt.

5.3.1 Zeichenketten aneinander hängen

Eine der häufigsten Operationen, die mit Strings durchgeführt werden, ist die Verkettung, d. h. das Aneinanderhängen mehrerer Zeichenketten. Hierfür können in Visual Basic sowohl das Pluszeichen (+) als auch das kaufmännische Und-Zeichen (&) verwendet werden.

Allerdings ist das kaufmännische Und (&) speziell für die Verkettung von Zeichenketten gedacht. Es kann nicht für Rechenoperationen verwendet werden. Hingegen ist das Pluszeichen ein normaler Rechenoperator, der in Abhängigkeit der Operanden entweder eine Addition ausführt, oder eben das Verketteten von Zeichenketten, falls beide Operanden Zeichenketten sind.

Verwenden Sie für die Verkettung von Zeichenketten immer das kaufmännische Und (&), denn dadurch ist Visual Basic immer in der Lage notfalls eine automatische Konvertierung einer Teilvariablen vorzunehmen, falls der Datentyp nicht *String* ist.



```
String1 & String2 [& String3] ...
```

Syntax

Folgendes Beispielprogramm zeigt die Funktion des Operators:

```
Private Sub Form_Load()  
Dim strTeil1 As String  
Dim strTeil2 As String  
Dim strverkettet As String  
strTeil1 = " Verketteter "  
strTeil2 = "String"  
strverkettet = strTeil1 & strTeil2  
MsgBox 1 & strverkettet  
End Sub
```



Zunächst werden drei Variablen des Datentyps *String* deklariert. Zwei von ihnen werden mit einem Zeichenkettenliteral initialisiert. Anschließend werden diese beiden Teilstrings durch den Operator & miteinander verbunden und der dritten Variablen zugewiesen.

In der Anweisung `MsgBox` schließlich wird die Variable *strverkettet* mit der numerischen Konstanten 1 verkettet. Hierbei findet eine automatische Konvertierung der numerischen Konstanten 1 statt, da der Operator dies erfordert.

Das Ergebnis dieses Programms mündet im Dialog aus Abbildung 5.26.

Abbildung 5.26:
Ausgabe eines ver-
ketteten Strings

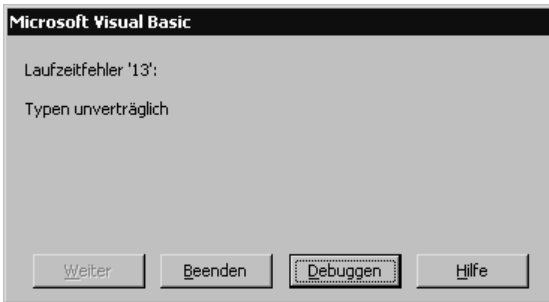


Der Unterschied zum Operator + wird deutlich, wenn Sie die Programmzeile der MsgBox-Funktion folgendermaßen ändern:

```
MsgBox 1 + strverkettet
```

Das Resultat dieses Aufrufs sehen Sie in Abbildung 5.27.

Abbildung 5.27:
Keine automatische
Konvertierung bei
Verwendung von +



Das Programm stürzt mit einem Laufzeitfehler ab. Die Interpretation der Zeile ergibt plötzlich eine Unverträglichkeit der Datentypen. Da der Plus Operator einer numerischen Konstante folgt, will Visual Basic eine Berechnung, also Addition, durchführen. Dies ist aber mit dem zweiten Operanden, einer Zeichenkette, nicht möglich.



Wie Sie an diesem Beispiel sehen, ist die Verwendung des speziellen Operators & für die Verkettung von Strings auf jeden Fall angeraten.

5.3.2 Länge einer Zeichenkette ermitteln

Die Länge eines Strings wird mit der Funktion *Len* ermittelt.

Syntax `Len(String)`

Das Ergebnis der Len-Anweisung ist ein Wert des Datentyps *Long*.

Folgende Programmzeilen demonstrieren die Funktionalität von *Len*.

```
Private Sub Form_Load()  
Dim lngLen As Long  
Dim strLen As String  
strLen = "Hallo"
```

```

IngLen = Len(strLen)
MsgBox "Das Wort " & strLen & " hat ein Länge von " & IngLen & "
Zeichen."
End Sub

```

Zunächst werden zwei Variablen deklariert. Die Variable *IngLen* wird verwendet, um das Ergebnis der *Len*-Funktion zu speichern. Die Variable *strLen* speichert ein Zeichenkettenliteral, dessen Länge ermittelt wird.

Die Ausgabe dieser Prozedur wird in Abbildung 5.28 gezeigt.



Abbildung 5.28:
Die Länge von Hallo wurde ermittelt

5.3.3 Entfernen von vor- oder nachlaufenden Leerstellen

Die Arbeit mit Zeichenketten erfordert sehr oft, diese von vor- oder nachlaufenden Leerzeichen zu befreien. Visual Basic kennt für diesen Zweck drei unterschiedliche Funktionen.

Die Funktion *RTrim* schneidet alle nachlaufenden Leerzeichen einer Zeichenkette, also rechtsbündig, ab.

```
RTrim(String)
```

Syntax

Das Gegenstück zu *RTrim* ist *LTrim*. Diese Funktion schneidet alle vorlaufenden Leerzeichen einer Zeichenkette, also linksbündig, ab.

```
LTrim(String)
```

Syntax

Die Funktion *Trim* schließlich ist eine Kombination der Funktionen *RTrim* und *LTrim*. Sie schneidet sowohl alle vorlaufenden als auch alle nachlaufenden Leerzeichen einer Zeichenkette, also beidseitig, ab.

```
Trim(String)
```

Syntax

Alle drei Funktionen liefern als Ergebnis einen String ohne die abgeschnittenen Leerzeichen.

Ein Beispiel für die Verwendung dieser Funktion bietet folgender Programmcode:

```

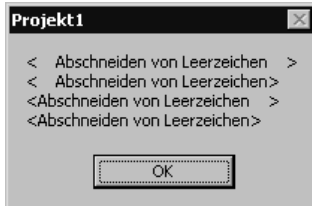
Dim strTrim As String
strTrim = "   Abschneiden von Leerzeichen   "
MsgBox "<" & strTrim & ">" & vbCrLf &
"<" & RTrim(strTrim) & ">" & vbCrLf &
"<" & LTrim(strTrim) & ">" & vbCrLf &
"<" & Trim(strTrim) & ">"

```



In Abbildung 5.29 können Sie die Ergebnisse der Funktionen begutachten. Damit die Leerzeichen in der Dialogbox auch »sichtbar« werden, wurden die einzelnen Teilstrings in die Zeichen < (kleiner) und > (größer) eingekleidet. Zudem wurde wiederum die Zeichenkettenkonstante `vbCrLf` verwendet um eine zeilenweise Ausgabe zu ermöglichen.

Abbildung 5.29:
Ergebnisse der
Trim-Funktionen



In der ersten Zeile des Dialogs wird der unveränderte, mit vor- und nachlaufenden Leerzeichen versehene String ausgegeben.

Die zweite Zeile zeigt den String, nachdem mit der Funktion `RTrim` die nachlaufenden Leerzeichen entfernt wurden.

Die dritte Zeile zeigt den String nach Verwendung der `LTrim`-Funktion.

In der vierten Zeile schließlich wurden die vor- und die nachlaufenden Leerzeichen mit der `Trim`-Funktion entfernt.

5.3.4 Umwandeln von Zeichenketten in Groß- oder Kleinbuchstaben

Manchmal ist es in Programmen notwendig, alle Zeichen einer Zeichenkette in Groß- oder Kleinbuchstaben umzuwandeln. Für diesen Zweck kennt Visual Basic zwei Funktionen.

Mit der Funktion `LCase` werden alle Zeichen einer Zeichenkette in Kleinbuchstaben umgewandelt.

Syntax `LCase(String)`

Die Funktion `UCase` erfüllt den gegenteiligen Zweck von `LCase`, wandelt also Kleinbuchstaben in Großbuchstaben um.

Syntax `UCase(String)`

Folgende Programmzeilen demonstrieren die Funktion von `LCase`.

```
Dim strKlein As String
strKlein = "Kleinbuchstaben"
MsgBox strKlein & vbCrLf & LCase(strGroßKlein)
```

Das Ergebnis dieser Programmzeilen sehen Sie in Abbildung 5.30.



Abbildung 5.30:
Umwandeln von
Groß- in
Kleinbuchstaben

In der ersten Zeile des Dialogs sehen Sie die Zeichenkette vor der Umwandlung. Die zweite Zeile zeigt die umgewandelte Zeichenkette. Es wurde lediglich der Anfangsbuchstabe der Wortes Kleinbuchstaben umgewandelt.

Die folgenden Programmzeilen zeigen die Funktionsweise von *UCase*.

```
Dim strGroß As String
strGroß = "Großbuchstaben"
MsgBox strGroß & vbCrLf & UCase(strGroß)
```

Das Ergebnis ist in Abbildung 5.31 zu sehen. Diesmal wurden alle Buchstaben bis auf den Anfangsbuchstaben umgewandelt.



Abbildung 5.31:
Umwandlung von
Klein- in
Großbuchstaben

Besonders zu beachten ist die Behandlung des Buchstabens »ß«. Dieser wurde bei der Umwandlung in Großbuchstaben nicht verändert. Die Funktionen *UCase* und *LCase* behandeln die länderspezifischen Sonderzeichen eines Zeichensatzes weitgehend korrekt.



5.3.5 Ausschneiden von Teilstrings

Unverzichtbar für das Programmieren mit Zeichenketten sind Funktionen für das Ausschneiden von Teil-Zeichenketten. Visual Basic kennt hierfür drei sehr komfortable Funktionen, die je nach Einsatzzweck wahlweise zu verwenden sind.

Die Funktion *Left* schneidet linksbündig aus. Sie benötigt dazu als Übergabeparameter den String, aus dem linksbündig ein Teilstring ausgeschnitten wird, und die Anzahl Zeichen, die ausgeschnitten werden sollen. Die Rückgabe der Funktion ist eine neue Zeichenkette.

```
Left(String, n)
```

Syntax

Die Funktion *Right* ist das entsprechende Gegenstück zu *Left*. Sie liefert als Ergebnis den rechtsbündigen Teil eines Strings. Übergabeparameter und Funktion sind ansonsten identisch.

Syntax `Right(String, n)`

Die Funktion *Mid* kann verwendet werden, wenn ein Teilstring aus der Mitte ausgeschnitten werden muss. Da die Startposition für das Ausschneiden nicht durch die Funktion selbst vorgegeben ist wie bei *Left* oder *Right*, muss diese als Übergabeparameter beim Aufruf angegeben werden.

Syntax `Mid$(String, Start[, Anzahl])`

Der Parameter *Start* bestimmt die Startposition des auszuschneidenden Teilstrings innerhalb von *String*.

Durch den Parameter *Anzahl* wird festgelegt, wie viele Zeichen ab dieser Position auszuschneiden sind. Er ist optional. Wird er weggelassen, werden alle Zeichen ab der Startposition bis zum Ende des Strings zurückgegeben.

Folgendes Beispiel zeigt die Verwendung von *Left*:



```
Dim strAusschneiden As String
strAusschneiden = "1234567890"
MsgBox strAusschneiden & vbCrLf & Left(strAusschneiden, 5)
```

Das Ergebnis dieses Programms sehen Sie in Abbildung 5.32.

Abbildung 5.32:
Ausschneiden
einer Zeichenkette
mit *Left*



In der ersten Zeile wurde der Originalstring ausgegeben. In der zweiten das Ergebnis, wenn mit der Funktion *Left* fünf Zeichen ausgeschnitten wurden. Der Originalstring bleibt von dieser Operation unberührt.



Falls Sie bereits mit anderen Programmiersprachen in Berührung gekommen sind, sei an dieser Stelle darauf hingewiesen, dass Visual Basic die Zählung der Zeichen einer Zeichenkette immer bei eins beginnt. In C oder Java beispielsweise beginnt die Zählung unglücklicherweise bei null.

Folgendes Beispiel zeigt die Verwendung von *Right*:



```
Dim strAusschneiden As String
strAusschneiden = "1234567890"
MsgBox strAusschneiden & vbCrLf & Right(strAusschneiden, 5)
```

Geändert wurde lediglich der Funktionsaufruf *Left* innerhalb der Funktion *MsgBox*. Es wird jetzt stattdessen die Funktion *Right* verwendet. Der entsprechende Ausgabedialog ist in Abbildung 5.33 zu sehen.



Abbildung 5.33:
Ausschneiden einer
Zeichenkette mit
Right

Zu guter Letzt wird mit folgenden Programmzeilen die Funktion von *Mid* gezeigt.

```
Dim strAusschneiden As String
strAusschneiden = "1234567890"
MsgBox strAusschneiden & vbCrLf & Mid(strAusschneiden, 3, 5)
```

Diesmal musste nicht nur das Funktions-Schlüsselwort geändert werden, sondern auch die Parameterliste. Denn die Funktion *Mid* benötigt im Gegensatz zu *Left* und *Right*, deren Startposition feststeht, die Position, ab welcher ausgeschnitten werden soll.

In Abbildung 5.43 können Sie sehen, dass auch bei *Mid* die Zählweise bei eins beginnt, und die Startposition wird dem Ergebnis hinzugefügt.



Abbildung 5.34:
Ausschneiden einer
Zeichenkette mit
Mid

5.3.6 Übung: Ausschneiden aus einer Zeichenkette

Lassen Sie den Anwender ein Wort eingeben und geben Sie ihm die Möglichkeit dieses Wort auf folgende 39 Arten auszugeben.

- ▶ Ausgabe der einzelnen Buchstaben in eine Liste.
- ▶ Ausgabe des ersten Buchstabens, der ersten zwei Buchstaben, der ersten drei Buchstaben usw. in einer Liste.
- ▶ Geben Sie das Wort normal aus, dann schneiden Sie bei jeder folgenden Ausgabe den zuvor links stehenden Buchstaben ab.



Lösung

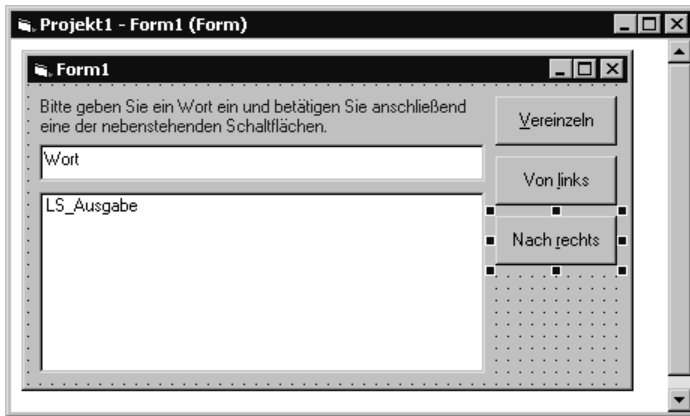
Zunächst benötigen wir eine Programmoberfläche, die alle Aufgaben erledigen kann. Die einzelnen notwendigen Elemente sind eine *TextBox* für die Eingabe des Wortes durch den Benutzer, eine *ListBox* für die Ausgabe des Ergebnisses und drei Schaltflächen für die jeweiligen Teilübungen.

Die Oberfläche in Abbildung 5.35 wurde lediglich noch um einen beschreibenden Text erweitert, der den Anwender auffordert ein Wort einzugeben



In Abbildung 5.35 wurde ein beschreibender Text als Bedienerführung und -info hinzugefügt. Die Vordergrundfarbe des Textes wurde auf Blau gesetzt, da ansonsten der Text im Rest des großen Fensters für das Auge des Anwenders untergeht. Wenn Sie also in einem Programmfenster nur eine oder zwei Informationen hervorheben möchten, so ist der Einsatz von Farbe dafür bestens geeignet.

Abbildung 5.35:
Programmier-
fläche mit Farbe



Die Lösung dieser Übung ist recht einfach, wenn man erkennt, dass jede Einzelübung durch den Einsatz einer *For...Next*-Schleife aufzulösen ist. Ansonsten werden die Funktionen *Mid*, *Left*, *Right* und *Len* benötigt.

aus Mitte ausschneiden

Für die erste Teilübung müssen aus dem eingegebenen Wort Einzelbuchstaben aus der Mitte einer vorgegebenen Zeichenkette geschnitten werden. Es gibt nur eine Funktion, die dazu direkt in der Lage ist, nämlich *Mid*.

Um das jeweils nächste Zeichen aus dem eingegebenen Wort auszuschneiden, muss lediglich der Parameter *Start* immer um eins erhöht werden. Diese Aufgabe lassen wir von *For...Next*-Schleife erledigen.

Die Länge der gelesenen Zeichen beträgt immer eins. Daher kann in der *Mid*-Funktion innerhalb der Schleife dieser Parameter mit einem numerischen Literal eingetragen werden.

Die Ereignisprozedur der Schaltfläche *Vereinzeln* sieht also aus wie folgt:

```
Private Sub BT_EinzelN_Click()  
Dim i As Integer  
For i = 1 To Len(TX_Eingabe.Text)  
    LS_Ausgabe.AddItem Mid(TX_Eingabe.Text, i, 1)  
Next i  
End Sub
```


In der ersten Zeile der Prozedur wird die Laufvariable der *For...Next*-Schleife deklariert. Diese wird innerhalb der *For...Next*-Schleife als Variable für den Parameter *Start* der *Mid*-Funktion verwendet.

Das Ende der Schleife wird ermittelt, indem die Länge des eingegebenen Wortes über die Funktion *Len* ermittelt wird.

Da die *Mid*-Funktion immer eine Länge von eins ausschneidet, wurden somit die Zeichen des Wortes vereinzelt. Diese müssen dann lediglich noch über die Methode *AddItem* in die Liste eingetragen werden.

Wenn Sie die Schaltfläche des Programms betätigen, erhalten Sie die Listeneinträge, die in Abbildung 5.36 zu sehen sind.

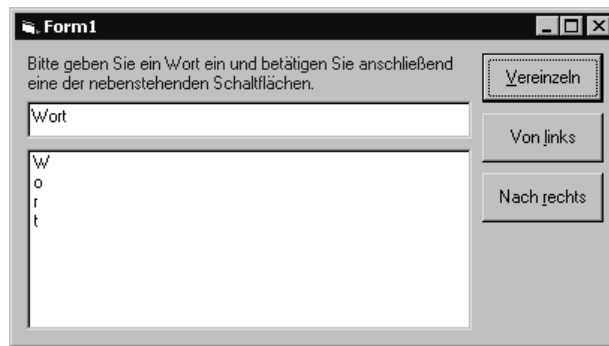


Abbildung 5.36:
Die Zeichen des
Worts wurden
vereinzelt

Der zweite Teil der Übung ist nach der Lösung des ersten beinahe schon erledigt. Die prinzipielle Struktur kann beibehalten werden. Allerdings werden in diesem Fall nicht einzelne Zeichen benötigt, sondern der auszugebende String wächst von links her an.

Die geeignete Funktion hierfür ist *Left*. Diese schneidet eine Anzahl Zeichen aus einer Zeichenkette und beginnt von links.

Die Länge der auszuschneidenden Zeichen erhöht sich wiederum bei jedem Durchgang um eins. Somit kann hier eine *For...Next*-Schleife zum Einsatz kommen.

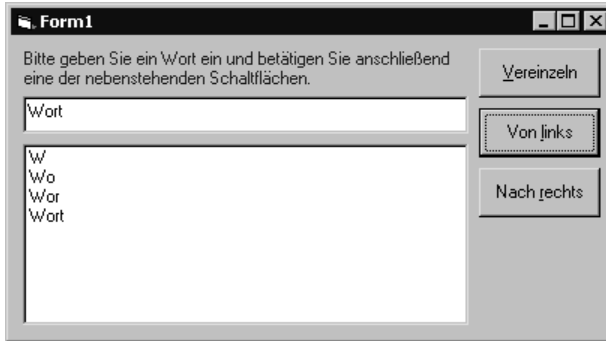
Die Ereignisprozedur der Schaltfläche *Von links* können Sie im Folgenden nachlesen:

```
Private Sub BT_Links_Click()
    Dim i As Integer
    For i = 1 To Len(TX_Eingabe.Text)
        LS_Ausgabe.AddItem Left(TX_Eingabe.Text, i)
    Next i
End Sub
```

Auch diesmal wird der Endwert der Zählvariablen mit der *Len*-Funktion definiert. Die Zählvariable wird in der *Left*-Funktion innerhalb der Schleife verwendet um die Länge des auszuschneidenden Textes zu ermitteln.

Die aus dieser Prozedur resultierenden Listeneinträge sehen Sie in Abbildung 5.37.

Abbildung 5.37:
Die Zeichen des
Wortes von links
erweiternd



Left oder Right?

Die letzte Teilübung ist ein klein wenig schwieriger. Vielleicht führte Sie die Formulierung der Übung in die Irre? Dort wird beschrieben, dass bei jedem Durchgang das erste Zeichen, also das linke, abgeschnitten wird. Was allerdings als Ergebnis für die Liste benötigt wird, sind die verbliebenen rechten Zeichen des Wortes. Es ist also trotz der Aufgabenformulierung ein Fall für die Funktion *Right*.

Das nächste Problem ist die Formulierung der For...Next-Schleifenbedingung. Wenn Sie zuerst das ganze Wort ausgeben und am Ende nur noch ein Zeichen, so muss die Schleife mit einer negativen Schrittweite laufen.

Der Rest ist an sich ein Klacks. Die Ereignisprozedur der Schaltfläche *Nach rechts* sehen Sie in folgenden Zeilen:

```
Private Sub BT_Rechts_Click()
    Dim i As Integer
    For i = Len(TX_Eingabe.Text) To 1 Step -1
        LS_Ausgabe.AddItem Right(TX_Eingabe.Text, i)
    Next i
End Sub
```

Die Zählvariable wird mit der Funktion *Len* bei ihrem ersten Durchlauf auf die Länge des Wortes initialisiert. Beim ersten Schleifendurchlauf wird also das ganze Wort ausgegeben. Bei jedem weiteren Schleifendurchlauf wird der Zählvariablen ein Zähler abgezogen, so dass jedes Mal ein Zeichen weniger von der Funktion *Right* zurückgegeben wird.

Die Ausgabe dieser Ereignisprozedur sehen Sie in Abbildung 5.38.

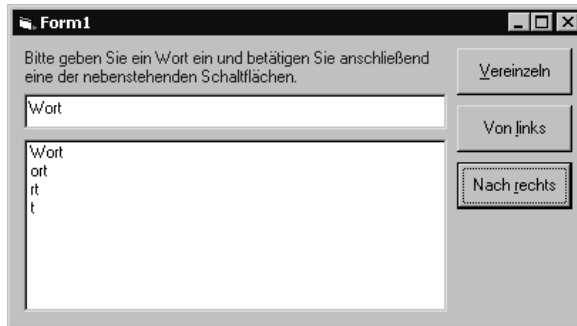


Abbildung 5.38:
Die Zeichen des
Wortes nach rechts
abnehmend

5.3.7 Ersetzen von Teilstrings

Eine zusätzliche Einsatzmöglichkeit ergibt sich, wenn die Funktion *Mid* innerhalb einer Zuweisung verwendet wird. In diesem Fall wird ein Teil des übergebenen Strings verändert bzw. ersetzt durch die Zuweisung.

`Mid$(String1, Start [, Anzahl]) = String2`

Die Zeichenkette *String2* wird durch diese Programmzeile in *String1* eingefügt. Die in *String1* vorhandenen Zeichen werden dabei überschrieben. Die Position, ab welcher *String1* überschrieben wird, wird durch den Parameter *Start* angegeben.

Der Parameter *Anzahl* ist optional. Wenn er weggelassen wird, wird die komplette Zeichenkette *String2* ab der Position *Start* eingefügt. Ist der Parameter *Anzahl* angegeben, so bezeichnet er die Anzahl Zeichen, die aus *String2* nach *String1* übernommen werden. Wird fälschlicherweise in *Anzahl* ein Wert angegeben, der die Länge von *String2* überschreitet, so stellt Visual Basic dies automatisch fest und übernimmt nur die tatsächlich vorhandenen Zeichen.

Da die Funktion *Mid* die Zeichen nicht tatsächlich in einen String einfügt, sondern die vorhandenen überschreibt, ist das Ersetzen eines Wortes nicht immer hiermit möglich. Ist beispielsweise das neue Wort länger, so muss der String über die Funktionen *Left* und *Right* in zwei Hälften geteilt und anschließend mit dem neuen Wort wieder verkettet werden.

Folgendes Beispiel zeigt die Funktionsweise von *Mid* in einer Zuweisung:

```
Dim strErsetzen As String
strErsetzen = "Sie müssen die Funktion Mid für das Ersetzen von
Zeichen verwenden."
MsgBox strErsetzen
Mid(strErsetzen, 5, 6) = "könnenx"
MsgBox strErsetzen
```

Abbildung 5.39 zeigt die Ausgabe des unveränderten Strings. Es dient lediglich der Kontrolle bzw. dem Vergleich mit dem Ergebnis der Zuweisung.

**Mid in einer
Zuweisung**

Syntax



Abbildung 5.39:
Ausgabe des unveränderten Strings



Abbildung 5.40:
Ausgabe des veränderten Strings



Abbildung 5.40 zeigt das Ergebnis der Zuweisung. Die Funktion *Mid* hat ab der fünften Stelle des Originalstrings sechs Zeichen ersetzt. Dabei wurden die ersten sechs Zeichen des Strings *könnenx* verwendet.

5.3.8 Suchen von Teilstrings in Strings

Sehr oft haben Sie in einem Programm die Aufgabe in einer Zeichenkette nach dem Vorhandensein einer anderen Zeichenkette zu suchen. Für genau diesen Zweck kennt Visual Basic die *Instr*-Funktion. Sie gibt die Position eines Strings in einem anderen, größeren String zurück.

Syntax `Instr([Start], Gesamtstring, Teilstring, [Vergleich])`

Wo wird gesucht? Der Parameter *GesamtString* gibt die Zeichenkette an, in welcher gesucht wird. Der Parameter *Teilstring* wiederum gibt die Zeichenkette an, nach der in *Gesamtstring* gesucht wird.

Ab welcher Stelle? Der Parameter *Start* bezeichnet die Position in *Gesamtstring*, ab welcher nach *Teilstring* gesucht wird. *Start* ist optional. Wird er nicht angegeben, so wird ab der ersten Stelle von *Gesamtstring* gesucht.

Was wird gesucht? Notwendig ist der Parameter *Start* immer dann, wenn in einem String mehrere Vorkommen einer Zeichenkette gesucht werden. Um das zweite Vorkommen des Teilstrings zu finden, würde in einer zweiten *Instr*-Anweisung die Startposition der erneuten Suche hinter das erste Vorkommen des Teilstrings gesetzt werden.

Wie wird gesucht? Mit dem Parameter *Vergleich* kann die Vergleichsmethode der *Instr*-Funktion eingestellt werden. Die möglichen Werte dieses Parameters sind:

- ▶ **VbBinaryCompare:** Die *Instr*-Funktion führt in diesem Fall einen *binären* Vergleich durch, der eine Unterscheidung zwischen Groß- und Kleinbuchstaben macht. Dies ist die Standardeinstellung, falls der Parameter weggelassen wird.

- ▶ **VbTextCompare:** Es wird ein *textbasierter* Vergleich durchgeführt. Bei dieser Art des Vergleichs wird keine Unterscheidung zwischen Groß- und Kleinbuchstaben gemacht. Wird beispielsweise der Kleinbuchstabe »a« in einem String gesucht, werden auch die im Gesamtstring vorkommenden Großbuchstaben »A« gefunden.
- ▶ **VbDatabaseCompare:** Diese Vergleichsmethode ist nur in Verbindung mit Daten einer Microsoft Access Datenbank verwendbar.

Folgender Programmcode zeigt die Verwendung der Instr-Funktion und demonstriert den Unterschied zwischen einem binären und einem textbasierten Vergleich:

```
Dim strSuchen As String
strSuchen = "Suchen ist schön."
MsgBox "Gefunden an " & Instr(strSuchen, "s") & ".ter Position."
MsgBox "Gefunden an " & Instr(1, strSuchen, "s", vbTextCompare) &
".ter Position."
```

Gesucht wird nach dem Vorkommen des Kleinbuchstabens »s« im String »Suchen mit Instr«. Bei der ersten Anweisung *Instr* wird der Parameter *Start* nicht angegeben, die Funktion *Instr* sucht daher ab der ersten Stelle des Gesamtstrings.

Abbildung 5.41 zeigt die Ausgabe des ersten Dialogs.

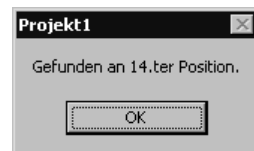


Abbildung 5.41:
Position bei
Verwendung des
binären Vergleichs

Da auch die Vergleichsoption weggelassen wurde, verwendet die *Instr*-Funktion einen *binären* Vergleich. Das kleine »s« wird an 14.ter Stelle im Gesamtstring gefunden. Es handelt sich um das *s* des Wortes *Instr*.

Die zweite Instr-Anweisung verwendet den textbasierten Vergleich. Damit dies geschieht, wurde im Parameter *Vergleich* die Konstante *vbTextCompare* eingetragen.

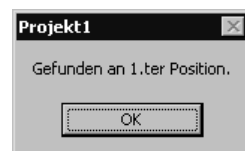


Abbildung 5.42:
Position bei einem
textbasierten
Vergleich

Abbildung 5.42 zeigt die Ausgabe des zweiten Dialogs. Das Programm findet den Kleinbuchstaben »s« an erster Position im String. Es handelt sich dabei um das *s* des Wortes *Suchen*. Es wurde also bei der Suche keine Rücksicht auf Groß- oder Kleinschreibung genommen.



5.3.9 Übung: Suchen und Ersetzen von Zeichen

Lassen Sie den Anwender einen Satz eingeben. Mit einer Funktion kann der Anwender die im Satz vorhandenen Leerzeichen zählen. Mit einer zweiten Funktion kann er alle Leerzeichen durch ein Minus-Zeichen ersetzen. Verwenden Sie die *Instr*-Funktion für das Finden der Leerzeichen und die *Mid*-Funktion für das Ersetzen derselben.

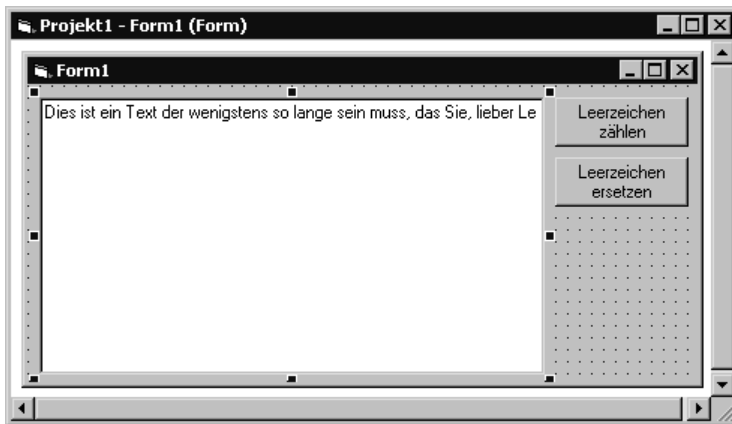
Lösung

Die Gestaltung der Oberfläche ist auch diesmal recht einfach. Sie benötigen eine Eingabemöglichkeit für den Anwender und zwei Schaltflächen um die Teile der Übung zu lösen.

Da bei dieser Übung allerdings ein ganzer Satz eingegeben werden soll, ist eine *TextBox* mit Standardeinstellungen nur ungenügend. Abbildung 5.43 zeigt Ihnen die Ausgabe eines längeren Textes in einer solchen.

Der Text wird nicht umgebrochen, sondern über den rechten Rand der *TextBox* hinausgeschrieben.

Abbildung 5.43:
TextBox mit Standardeinstellungen



In Abbildung 5.44 hingegen können Sie den ganzen Text lesen, da dieser umgebrochen wurde.



Wenn Sie längere Texte eingeben müssen, können Sie die Eigenschaft *MultiLine* der *TextBox* auf den Wert *True* setzen. Danach wird der Text in der *TextBox* automatisch umgebrochen, sobald der rechte Rand erreicht wird. Sind im eingegebenen Text Leerzeichen vorhanden, werden diese für den Umbruch verwendet, so dass ganze Wörter nicht auseinander gerissen werden.

Für die erste Teilübung wird die Ereignisprozedur der Schaltfläche *Leerzeichen zählen* verwendet. Mein Vorschlag für den Programmcode sehen Sie in folgenden Zeilen:

```

Private Sub BT_Zählen_Click()
Dim intAnzahl As Integer
Dim intPosition As Integer
Dim keinemehrda As Boolean
keinemehrda = False
intAnzahl = 0
intPosition = 1
Do
    intPosition = InStr(intPosition, TX_Eingabe.Text, " ")
    If intPosition > 0 Then
        intAnzahl = intAnzahl + 1
        intPosition = intPosition + 1
    Else
        keinemehrda = True
    End If
Loop Until keinemehrda
MsgBox "Es wurden " & intAnzahl & " Leerzeichen gefunden"
End Sub

```



Abbildung 5.44:
 TextBox mit umgebrochenem Text,
 Eigenschaft
 Multiline

Es werden drei Variablen benötigt. Die Variable *intAnzahl* wird verwendet, um die Leerzeichen zu zählen, sie wird daher auf 0 initialisiert.

Die Variable *intPosition* wird verwendet, um die Startposition der *Instr*-Funktion festzulegen, und sie speichert gleichzeitig die gefundene Position innerhalb des Strings. Sie muss mit dem Wert 1 initialisiert werden, da für die Startposition nur Werte größer 0 gültig sind.

Die dritte Variable heißt *keinemehrda*. Sie wird als Abbruchkriterium für die *Do...Loop*-Schleife verwendet. Da diese mit dem Schlüsselwort *Until* arbeitet, muss die Variable auf *False* initialisiert werden.

Die *Instr*-Funktion liefert als Ergebnis die Position des gesuchten Zeichens. Wird das Zeichen nicht gefunden, so liefert sie den Wert 0 zurück.

Die *If*-Abfrage testet, ob das Zeichen gefunden wurde. Ist dies der Fall, so wird der Zähler für die gefundenen Zeichen um eins erhöht.

**Start muss größer
 0 sein**

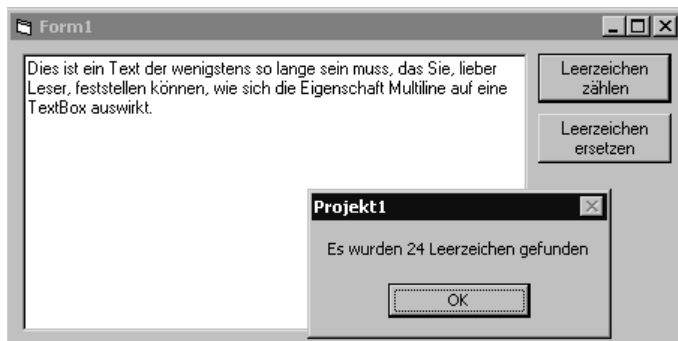


Wurde ein Zeichen gefunden, so muss die Startposition für den nächsten *Instr*-Aufruf um eins erhöht werden. Dies ist sehr wichtig. Tun Sie es nicht, wird die Schleife nie beendet, da die *Instr*-Funktion immer das gleiche Zeichen findet. Ihr Programm befindet sich dann in einer Endlosschleife.

Wurde das gesuchte Zeichen nicht gefunden, so wird die Abbruchbedingung auf *True* gesetzt. Die Schleife wird sofort beendet und die gezählten Zeichen werden ausgegeben.

In Abbildung 5.45 können Sie sehen, dass die Prozedur funktioniert.

Abbildung 5.45:
Leerzeichen zählen



Die nächste Teilübung ist im Grunde nicht schwieriger als die erste. Wir können sogar die Grundstruktur der ersten Teilübung wiederverwenden, wie Sie in folgenden Programmzeilen ersehen können:

```
Private Sub BT_Ersetzen_Click()
    Dim strArbeitsString As String
    Dim intPosition As Integer
    Dim keinemehrda As Boolean
    keinemehrda = False
    intPosition = 1
    strArbeitsString = TX_Eingabe.Text
    Do
        intPosition = Instr(strArbeitsString, " ")
        If intPosition > 0 Then
            Mid(strArbeitsString, intPosition, 1) = "-"
        Else
            keinemehrda = True
        End If
    Loop Until keinemehrda
    TX_Eingabe.Text = strArbeitsString
End Sub
```

Für diese Übung benötigen wir die Variable *intAnzahl* nicht. Sie wurde gestrichen.

Da wir den Text jedoch diesmal ändern möchten, benötigen wir stattdessen eine Variable, die den Text für die Änderungen zwischenspeichert. Es ist nicht möglich, die Funktion *Mid* in einer Zuweisung direkt auf die Eigenschaft *Text* einer *TextBox* anzuwenden.



Die Hilfsvariable *strArbeitsString* wird daher vor der Schleife mit dem Text aus der *TextBox* initialisiert. Innerhalb der Schleife wird nur mit der Hilfsvariablen gearbeitet.

Für die *Instr*-Funktion benötigen wir diesmal keine Startposition. Da wir jedes gefundene Zeichen sofort verändern, besteht keine Gefahr, ein Zeichen zweimal zu finden.

Suchen

Wird nun ein Leerzeichen gefunden, so wird es über die *Mid*-Zuweisung im Hilfsstring durch ein Minus-Zeichen ersetzt.

Ersetzen

Sobald keine Leerzeichen mehr im Text vorhanden sind, wird die Schleife beendet. Der Hilfsstring wird der Eigenschaft *Text* der *TextBox* zugewiesen und die Übung ist abgeschlossen.

Die Ausgabe dieser Prozedur können Sie in Abbildung 5.46 betrachten.

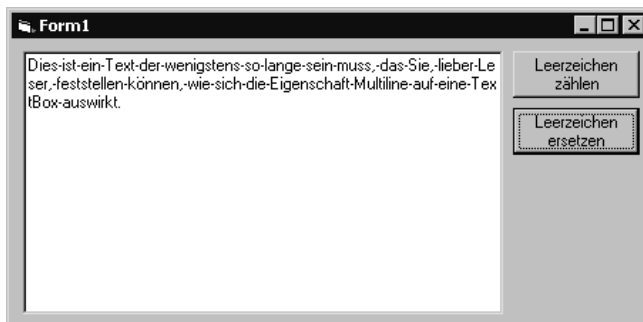


Abbildung 5.46:
Leerzeichen
wurden ersetzt

Beachten Sie bitte den Zeilenumbruch in der *TextBox*. Jetzt, nachdem keine Leerzeichen mehr vorhanden sind, wird der Text einfach mit dem Zeichen umbrochen, welches den rechten Rand überschreiten würde.



5.3.10 Übung: Textdrehen

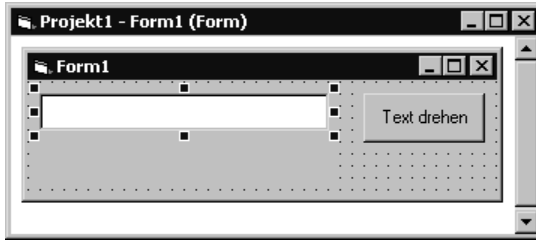
Lassen Sie den Anwender ein Wort eingeben. Geben Sie dieses Wort rückwärts aus. Sorgen Sie dafür, dass bei dem rückwärts ausgegebenen Wort der erste Buchstabe großgeschrieben wird und der Rest klein.



Lösung

Die Oberfläche für diese Übung ist unspektakulär. Wir benötigen eine *TextBox* für die Eingabe des Wortes, ein *Label* für die Ausgabe des umgedrehten Wortes und eine *Schaltfläche* um die Funktion aufzurufen. Sie können diese Oberfläche in Abbildung 5.47 betrachten.

Abbildung 5.47:
Oberfläche der
Übung Textdrehen



Um ein Wort umzudrehen, müssen die Zeichen des Wortes vereinzelt werden. In einer früheren Übung haben wir dies bereits durchgeführt. Diesmal muss allerdings die Vereinzelnung beim letzten Zeichen beginnen und beim ersten enden. Für diese Übung können wir die Funktionen *Mid* und *Len* und eine *For...Next*-Schleife verwenden.

Um die Groß- und Kleinschreibung aufgabengerecht zu machen, werden zusätzlich die Funktionen *UCase* und *LCase* benötigt.

Mein Vorschlag ist folgender Programmcode:

```
Private Sub BT_Drehen_Click()
    Dim i As Integer
    Dim gedrehterText As String
    gedrehterText = ""
    For i = Len(TX_Eingabe.Text) To 1 Step -1
        If i < Len(TX_Eingabe.Text) Then
            gedrehterText = gedrehterText & LCase(Mid(TX_Eingabe.Text, i,
1))
        Else
            gedrehterText = gedrehterText & UCase(Mid(TX_Eingabe.Text, i,
1))
        End If
    Next i
    LB_Ausgabe.Caption = gedrehterText
End Sub
```

In der Hilfsvariablen *gedrehterText* wird innerhalb der *For...Next*-Schleife das Ergebnis zusammengebaut.

Die *For...Next*-Schleife selbst läuft wie besprochen rückwärts, vom letzten Zeichen des Wortes bis zum ersten. Die *Mid*-Anweisungen vereinzeln jeweils das Zeichen, auf dem die Zählvariable momentan steht.

Die *If*-Struktur prüft, ob es sich um das letzte Zeichen des Originalstrings handelt. Ist dies der Fall, so wird das Zeichen mit der Funktion *UCase* in einen Großbuchstaben umgewandelt, ansonsten wird die Funktion *LCase* verwendet, um einen Kleinbuchstaben zu erhalten.

Nach Abschluss der *For...Next*-Schleife wird das Ergebnis in der Variablen *gedrehterText* im Label zur Anzeige gebracht.

Das Ergebnis dieser Bemühungen können Sie in Abbildung 5.48 bewundern.



Abbildung 5.48:
Der Text wurde
gedreht

5.3.11 Umwandlung einer Zeichenkette in eine Zahl

Im Abschnitt über die Funktion *InputBox* haben wir bereits festgestellt, dass es manchmal notwendig ist, eine Zeichenkette in einen numerischen Wert umzuwandeln.

Visual Basic stellt hierfür die Funktion *Val*, zur Verfügung. Diese wandelt einen String in einen geeigneten numerischen Wert um.

`Val$(Zahlstring)`

Die *Val*-Funktion interpretiert den Übergabeparameter *Zahlstring* von links nach rechts. Sobald ein Zeichen auftaucht, welches nicht mehr als Zahl interpretierbar ist, endet die Konvertierung und das Ergebnis wird zurückgeliefert. Leerzeichen werden bei der Konvertierung ignoriert.

Als Dezimaltrennzeichen wird der Punkt gewertet. Länderspezifische Abweichungen, wie die Verwendung des Kommas als Dezimaltrennzeichen, werden nicht berücksichtigt.

Es wird jeweils nur das erste Vorkommen des Punkts als Dezimaltrenner interpretiert. Ein weiterer Punkt in *Zahlstring* wird als nicht interpretierbares Zeichen gewertet, und somit ist die Konvertierung an dieser Stelle abgeschlossen.

Folgendes Beispiel demonstriert die Funktionsweise von *Val*:

```
MsgBox Val(" 1234.34 5")
MsgBox Val(" 1234,34 5")
```

Abbildung 5.49 zeigt die Ausgabe des ersten Dialogs. Das Ergebnis ist 1234,345. Sowohl die vorlaufenden als auch das Leerzeichen zwischen 4 und 5 wurden von *Val* ignoriert.

Syntax

Dezimaltrennzeichen nicht länderspezifisch!

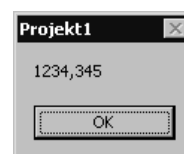


Abbildung 5.49:
Konvertierung mit
Dezimaltrennzeichen



Es ist auffallend, dass bei der Ausgabe der Zahl statt des Punktes als Dezimaltrennzeichen ein Komma eingefügt wurde. Der Grund hierfür ist in den länderspezifischen Einstellungen des Rechners zu suchen. Da dort als Dezimaltrennzeichen ein Komma hinterlegt ist, führt die automatische Umwandlung der Zahl in einen String im Aufruf der Funktion *MsgBox* zum Einfügen des Kommas.

Im zweiten Dialog wurde statt des Punktes ein Komma eingefügt. In Abbildung 5.50 sehen Sie, dass dies nicht als Dezimaltrennzeichen, sondern als nicht konvertierbar gewertet wurde.

Abbildung 5.50:
Falsches Dezimaltrennzeichen



5.3.12 Umwandlung einer Zahl in eine Zeichenkette

Die Funktion *Str* wandelt einen numerische Ausdruck gezielt in einen String um. Sie wird nicht sehr oft benötigt, da die Umwandlung einer Zahl in einen String von Visual Basic sehr oft automatisch erledigt wird.

Syntax `Str$(Ausdruck)`

Im Parameter *Ausdruck* wird der umzuwandelnde numerischer Wert übergeben. Handelt es sich dabei um eine Fließkommazahl, so muss als Dezimaltrennzeichen zwingend ein Punkt verwendet werden.

Folgende Programmzeile zeigt die Funktionsweise von *Str*.

```
MsgBox "<" & Str(1234.56) & ">"
```

Abbildung 5.51 zeigt den resultierenden Dialog.

Abbildung 5.51:
Konvertierung mit
Str-Funktion



Dezimaltrennzeichen nicht länderspezifisch!

Die Funktion *Str* verwendet im Ergebnis, ohne Rücksicht auf die länderspezifischen Einstellungen, den Punkt als Dezimaltrennzeichen.

Zudem wird bei der Konvertierung mittels *Str* der konvertierten Zahl ein Leerzeichen vorangestellt. Die *Str*-Funktion reserviert im umgewandelten String das erste Zeichen für ein Vorzeichen. Ist die Zahl positiv, so wird das Zeichen »+« weggelassen und durch ein Leerzeichen dargestellt.

5.3.13 Formatieren von Strings

Die Funktion *Format* ist ein Multitalent um nahezu beliebig formatierte Zeichenketten zu erzeugen. In ihrer einfachsten Form kann sie anstelle von *Str* verwendet werden.

`Format(Ausdruck [, Formatierung [, erster_tag[, erstewoche]])`

In der kürzesten Form, ohne alle optionalen Parameter, ergibt sich eine Umwandlung, die der Funktion *Str* in etwa entspricht. *Format* berücksichtigt allerdings bei der Konvertierung die länderspezifischen Einstellungen des Rechners und lässt im Gegensatz zu *Str* den Platzhalter für ein positives Vorzeichen weg. *Format* funktioniert wie die automatische Konvertierung von Visual Basic.

Die Funktion *Format* kann auch in Programmen verwendet werden, die international eingesetzt werden sollen. Die Verwendung der *Str*-Funktion ist nur dann anzuraten, wenn aus Kompatibilitätsgründen oder eventuell wegen des führenden Leerzeichens ein Vorteil zu erwarten ist.

Das ganze Potenzial der *Format*-Funktion wird allerdings erst ausgeschöpft, wenn mit dem Parameter *Formatierung* gearbeitet wird. Dieser Parameter stellt nämlich eine Schablone dar, anhand derer die Ausgabe formatiert wird.

Der Formatstring hat einen festgelegten Aufbau. Um ein vernünftiges Ergebnis zu erzielen, müssen Sie die Formatierungszeichen kennen, die verwendet werden dürfen.

Zum einen ermöglicht die Funktion *Format* die formatierte Ausgabe von numerischen Werten, zum anderen die von Datums- und Zeitangaben. Im Folgenden werden die hierfür notwendigen Formatierungszeichen jeweils in einer eigenen Tabelle gelistet.

Formatierungszeichen für numerische Werte

Tabelle 5.6 zeigt alle Formatierungszeichen, die zur Aufbereitung eines numerischen Wertes innerhalb des Formatierungsstrings erlaubt sind.

Zeichen	Bedeutung
.	Der Punkt wird verwendet um die Position des Dezimaltrennzeichens bzw. die Anzahl der Nachkommastellen festzulegen. Er darf nur einmal im Formatstring vorkommen. Im formatierten String erscheint an der entsprechenden Stelle das in den länderspezifischen Einstellungen festgelegte Zeichen für den Dezimaltrenner. Mit deutschen Ländereinstellungen ist dies z. B. das Komma. Achtung: Dieses Formatierungszeichen ist laut Visual Basic 6.0 Online-Hilfe abhängig von den länderspezifischen Einstellungen des Rechners.

Syntax

Format = Str ?



Tabelle 5.6:
Sonderzeichen zur
Umwandlung und
Formatierung einer
Zahl mittels *Format*

Zeichen	Bedeutung
,	Mit dem Komma wird angezeigt, dass ein Tausendertrennzeichen verwendet werden soll. Es kann sich einmalig an einer beliebigen Stelle im Vorkommabereich befinden, muss allerdings auf beiden Seiten eingeschlossen sein von Platzhaltern für Zahlen (# oder 0). Im formatierten String erscheint wie bei dem Formatierungszeichen Punkt das länderspezifische Tausendertrennzeichen. Mit deutschen Ländereinstellungen ist dies der Punkt.
#	Das Zeichen # steht jeweils für eine Ziffer. Im Vorkommabereich wird die Anzahl dieser Formatierungszeichen ignoriert, d.h. die Vorkommastellen der zu formatierenden Zahl werden ohne Berücksichtigung dieser Formatierungszeichen ausgegeben. Im Gegenzug hierzu wird im Nachkommabereich die Anzahl vorhandener Formatierungszeichen berücksichtigt. Hat die zu formatierende Zahl mehr Nachkommastellen, als Formatierungszeichen vorhanden sind, so erscheint die konvertierte Zahl im String mit der angegebenen Anzahl Nachkommastellen. Die Zahl wird dabei auf- oder abgerundet.
0	Die Null ist ein Platzhalter für eine Ziffer. Sie verhält sich analog zu '#', zeigt jedoch führende und nachstehende Nullen an, wenn die Anzahl Stellen kleiner sein sollte, als durch den Formatierungsstring vorgesehen.
Andere Zeichen oder Strings	Anzeige von Zeichenwerten. Alle anderen Zeichen im Formatstring werden nicht als Formatzeichen interpretiert, sondern direkt in den Ausgabestring übernommen.

Folgende Programmzeile zeigt die prinzipielle Funktionsweise des Formatstrings für numerische Werte.

```
MsgBox Format(1234.56, "#,#.##")
```

Ausgegeben wird der Dialog aus Abbildung 5.52.

Abbildung 5.52:
Eine formatierte
Zahl



Die Ausgabe erklärt sich anhand der Tabelle wie folgt. Vor dem Punkt im Formatstring wird durch das von den Zeichen # eingeschlossene Komma festgelegt, dass ein Tausendertrennzeichen dargestellt werden soll. Hinter dem Punkt wird festgelegt, wie viele Dezimalstellen angezeigt werden. In diesem Fall sind es zwei.

5.3.14 Übung: Formatierte Ausgabe von Zahlen

Schreiben Sie ein Programm, welches eine durch den Anwender eingegebene Zahl in drei verschiedenen Arten formatiert:

- ▶ Vor dem Komma ohne Tausendertrennzeichen, sechs Nachkommastellen.
- ▶ Vor dem Komma mit Tausendertrennzeichen, vier Nachkommastellen.
- ▶ Vor dem Komma bis 5 Stellen mit führenden Nullen, sechs Nachkommastellen.

Lösung

Für die Lösung dieser Übung benötigen wir drei Schaltflächen, eine TextBox und ein Label für die Ausgabe des formatierten Ergebnisses.

Meinen Vorschlag sehen Sie in Abbildung 5.53.

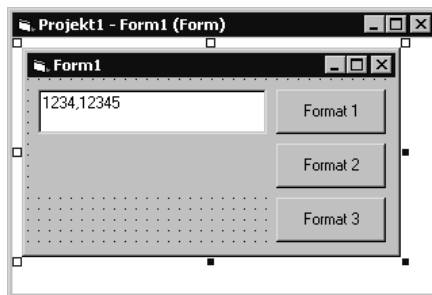


Abbildung 5.53:
Oberfläche Übung
Formatierte Aus-
gabe von Zahlen

Die erste Teilübung wird in der Ereignisroutine der Schaltfläche *Format 1* hinterlegt. Der Programmcode ist recht kurz, wie Sie in folgenden Zeilen sehen:

```
Private Sub Command1_Click()  
    LB_Ausgabe.Caption = Format(CDb1(TX_Eingabe.Text), ".#####")  
End Sub
```

Der Formatstring definiert eine Schablone, die maximal sechs Nachkommastellen darstellt. Vor dem Dezimaltrenner ist kein Formatzeichen notwendig. Abbildung 5.54 zeigt die Wirkung dieser Formatierung auf die Zahl *1234,12345*.

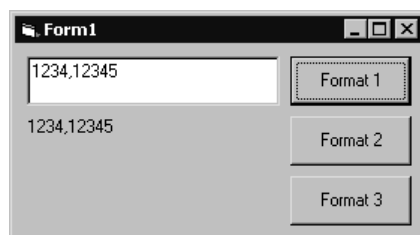


Abbildung 5.54:
Formatierung mit
sechs Nachkom-
mastellen



Da die zur Formatierung übergebene Zahl weniger Nachkommastellen hat, als im Formatstring angegeben sind, werden alle Nachkommastellen der Zahl ausgegeben. Es wird allerdings kein zusätzliches Zeichen für das sechste Zeichen des Formatstrings angehängt.

Auf- oder Abrundung

Würde die Zahl über sieben Nachkommastellen verfügen, so würde das siebte Zeichen abgeschnitten werden. Dabei findet dann eine Ab- bzw. Aufrundung an der sechsten Stelle statt.

Der Programmcode der zweiten Teilübung wurde in der Ereignisprozedur der Schaltfläche *Format 2* hinterlegt. Sie sehen diese in den folgenden Zeilen.

```
Private Sub Command2_Click()  
    LB_Ausgabe.Caption = Format(CDb1(TX_Eingabe.Text), "#,#.####")  
End Sub
```

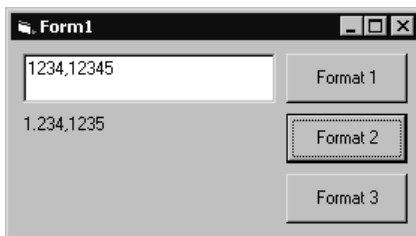
Im Formatstring sind nur vier Nachkommastellen definiert. Zudem wurde vor dem Dezimaltrennzeichen die Formatanweisung *#, #* angegeben. Das Komma steht in diesem Fall für ein Tausendertrennzeichen. Es kann an beliebiger Stelle vor dem Dezimaltrenner stehen, muss allerdings, um gültig zu sein, von den Formatzeichen *#* eingekleidet werden.



Die Formatzeichen *#* haben nur hinter dem Dezimaltrennzeichen die Bedeutung als Platzhalter für Zahlen. Vor dem Dezimaltrenner werden sie von Visual Basic ignoriert, außer sie werden benötigt um Tausendertrennzeichen zu definieren. Visual Basic gibt vor dem Dezimaltrenner immer die komplette Zahl aus.

Wie sich dieser Formatstring auf die gleiche Zahl auswirkt, sehen Sie in Abbildung 5.55.

Abbildung 5.55:
Formatierung mit
Tausendertrennzeichen



Nachkommastellen

Da die *Format*-Anweisung nur vier Nachkommastellen erlaubt, wird bei der Formatierung die fünfte Stelle abgeschnitten. Hierbei führt die fünf an der fünften Nachkommastelle zu einer Aufrundung der vierten Nachkommastelle.

Tausendertrennzeichen

Vor dem Dezimaltrenner wurde ein Tausendertrennzeichen zwischen die 1 und die 2 eingefügt.

Die Lösung der dritten Teilübung befindet sich in der Ereignisroutine der Schaltfläche *Format 3*. Der Programmcode ist in den folgenden Zeilen zu sehen:


```
Private Sub Command3_Click()
    LB_Ausgabe.Caption = Format(CDb1(TX_Eingabe.Text), "00000.000000")
End Sub
```

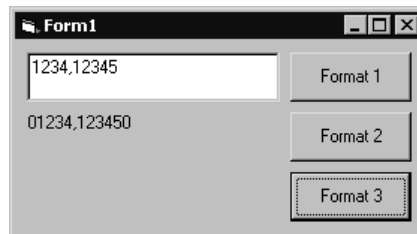
Diesmal wurden im Formatstring die Formatzeichen 0 verwendet. Diese sollen vor- und nachlaufende Nullen erzeugen.

Im Gegensatz zu den Formatzeichen # hat die Anzahl der Formatzeichen 0 vor dem Dezimaltrenner eine Bedeutung. Hat die Zahl weniger Stellen vor dem Komma, so werden diese mit Nullen aufgefüllt. Gleiches gilt für die Nachkommastellen.



Werden allerdings mehr Stellen vor dem Dezimalpunkt übergeben, so werden diese in das Ergebnis ausgegeben. Sind hinter dem Dezimalpunkt mehr Stellen vorhanden, so verhält sich das Formatzeichen 0 gleich wie das Formatzeichen #. Es werden, mit Auf- oder Abrundung, die überzähligen Stellen abgeschnitten.

Die Wirkung dieses Formatstrings auf unsere Testeingabe sehen Sie in Abbildung 5.56.



*Abbildung 5.56:
Formatieren mit
führenden und
nachlaufenden
Nullen*

Diese Formatierungsart hat zwei wichtige Anwendungen. Erstens kann diese Formatierung verwendet werden, um in einer Tabelle Zahlenwerte untereinander auszugeben und dabei den Dezimaltrenner immer in der gleichen Spalte zu haben.



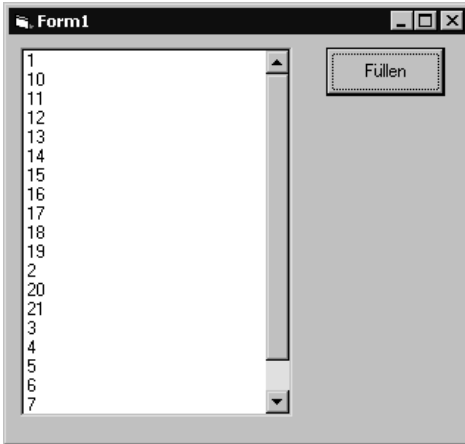
Zweitens ist es möglich, eine mit Zahlen sortierte Liste korrekt auszugeben. Eine sortierte Liste der Zahlen 1 bis 21 wird die Reihenfolge wie in Abbildung 5.57 haben.

Warum ist die Sortierung in Abbildung 5.57 falsch? Nun Sortierung ist tatsächlich korrekt. Denn die Liste enthält ja nicht tatsächlich Zahlen, sondern Zeichenketten. Bei einer Zeichenkette wird der Vergleich aber nur bis zum ersten unterschiedlichen Zeichen durchgeführt. Daher ist die Zeichenkette »2« größer als »10«.



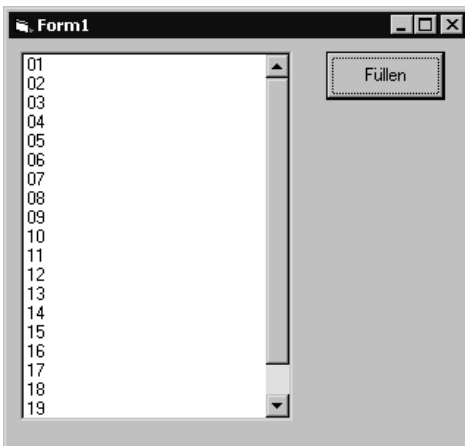
Dass die Sortierung der Strings auch korrekt ist im Sinne von numerischen Werten, kann über einen Trick erreicht werden. Die Zahlen werden über Format mit führenden Nullen versehen. Denn 02 ist in der Tat auch als Zeichenkette kleiner als 10.

Abbildung 5.57:
»Sortierte« Liste
von »Zahlen«



Die Sortierung, nachdem die Zahlen vor dem Eintrag in die Liste formatiert wurden, sehen Sie in Abbildung 5.58.

Abbildung 5.58:
Sortierte Liste von
Zeichenkette
bzw. Zahlen



Formatierungszeichen für Zeit- und Datumsangaben

In Tabelle 5.7 werden die Formatierungszeichen aufgeführt, die notwendig sind, um Zeit- und Datumsangaben zu formatieren.

Tabelle 5.7:
Datums- und
Zeitangaben
formatieren

Zeichen	Bedeutung
D	Anzeige des Tagesdatums ohne vorangestellte Null bei einstelligen Monatstagen
Dd	Datumsanzeige mit vorangestellten Nullen
Ddd	Anzeige des Tages in alphanummerischer Abkürzung
Dddd	Anzeige des Tages als Wochentag

Zeichen	Bedeutung
Dddd	Anzeige des kompletten Datums entsprechend den Systemeinstellungen für kurzes Datumsformat.
Dddddd	Anzeige des kompletten Datums entsprechend den Systemeinstellungen für das lange Datumsformat.
W	Zeigt den Wochentag als Zahl an, wobei bei Sonntag beginnend mit 1 gezählt wird bis 7 für Samstag.
Ww	Zeigt die Kalenderwoche an.
M	Anzeige der Monatszahl ohne vorangestellte Null
mm	Anzeige der Monatszahl mit vorangestellter Null
mmm	Anzeige des Monats in alphanummerischer Abkürzung
mmmmm	Anzeige des ausgeschriebenen Monatsnamens
yy	Anzeige einer zweistelligen Jahreszahl
yyyy	Anzeige einer vierstelligen Jahreszahl
h	Anzeige der Stunde ohne vorangestellte Null
hh	Anzeige der Stunde mit vorangestellter Null
m	Anzeige der Minute ohne vorangestellte Null
mm	Anzeige der Minute mit vorangestellter Null
s	Anzeige der Sekunde ohne vorangestellte Null
ss	Anzeige der Sekunde mit vorangestellter Null
:	Trennzeichen für Zeitangaben. Dient zur Trennung von Stunden, Minuten und Sekunden bei Zeitangaben.
/	Trennzeichen für Datumsangaben. Dient der Trennung von Tag, Monat und Jahr.

Die folgende Programmzeile zeigt die prinzipielle Funktionsweise der Formatierungszeichen aus Tabelle 5.7.

```
MsgBox Format(Now, "dd/mm/yyyy hh:mm:ss")
```

In Abbildung 5.59: Ausgabe eines formatierten Datums sehen Sie die Ausgabe des Dialogs.

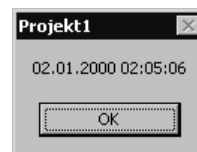


Abbildung 5.59: Ausgabe eines formatierten Datums

Mit der Funktion *Now* wird die aktuelle Systemzeit zur Formatierung übergeben. Der Formatstring sorgt dafür, dass jeweils zweistellig Tag und Monat, dann vierstellig das Jahr und getrennt durch ein Leerzeichen je zweistellig Stunden, Minuten und Sekunden im Ergebnis gezeigt werden.

**Funktion Now
für aktuelle
Systemzeit**

Durch die Angabe des Doppelpunktes und des Schrägstrichs wird die Format-Funktion dazu veranlasst, in den Ergebnisstring an den entsprechenden Stellen das landesspezifische Zeit- bzw. Datumstrennzeichen einzufügen.



5.3.15 Übung: Uhr

Programmieren Sie eine Uhr. Folgende Daten sollen angezeigt werden:

- ▶ Uhrzeit mit Sekunden
- ▶ Datum im langen Datumsformat
- ▶ die aktuelle Kalenderwoche.



Um diese Übung zu lösen sollten Sie sich mit dem Steuerelement *Timer* auseinander setzen.

Lösung

In der Oberfläche aus Abbildung 5.60 wurden für die drei geforderten Ausgaben drei verschiedene Steuerelemente des Typs Label verwendet. Nur dann können Sie für die Schriftgrößen unterschiedliche Werte einsetzen und somit eine Strukturierung der Oberfläche erreichen.

Abbildung 5.60:
Programmober-
fläche der
Übung Uhr



Das Steuerelement *Timer* ist als Icon auf der Programmoberfläche zu sehen. Zur Laufzeit des Programms ist dieses Icon nicht sichtbar. Der *Timer* produziert in zyklischen Abständen Ereignisse. Der Abstand zwischen diesen Ereignissen wird über die Eigenschaft *Interval* eingestellt (Abbildung 5.61).

sekundengenau?

Wenn Sie eine Uhrzeit sekundengenau darstellen möchten, darf das Interval auf jeden Fall nicht größer sein als eine Sekunde. Im Grunde haben wir bei einem Interval von einer Sekunde bereits eine max. Abweichung von +/- 999 Millisekunden gegenüber der Systemuhrzeit.



Abbildung 5.61:
Eigenschaft
Interval des
Timers einstellen

Die Darstellung der Zeit- und Datumsanzeigen wird in der Ereignisroutine *Timer* des *Timers* gemacht. Diese wird im Programmierer automatisch geöffnet, wenn Sie auf das Icon des *Timers* im *Designer* doppelklicken. Den notwendigen Programmcode können Sie in folgenden Zeilen sehen:

```
Private Sub Timer1_Timer()  
    LB_Datum.Caption = Format(Now, "dddddd")  
    LB_Uhrzeit.Caption = Format(Now, "HH:MM:SS")  
    LB_KW.Caption = "Kalenderwoche " & Format(Now, "WW")  
End Sub
```

Hätten Sie gedacht, dass es so einfach ist, eine Uhr zu programmieren?

Für die Ausgabe der unterschiedlichen Daten wird jeweils eine eigene Formatanweisung mit einem Formatstring verwendet. Es wird immer die aktuelle Uhrzeit formatiert, die über die Funktion *Now* ermittelt wird.

Der erste Formatstring "dddddd" formatiert das so genannte lange Datumsformat. Es handelt sich hierbei um ein Datum, in welchem der Wochentag und der Monat ausgeschrieben werden. Die Jahreszahl wird vierstellig angegeben. Das Ergebnis dieser Formatierung ist abhängig von den landesspezifischen Einstellungen des Rechners. Das heißt, das Format würde auf einem englischen Windows ein englisches Format anzeigen, auf einem spanischen Windows ein spanisches und auf einem deutschen eben ein deutsches Format.

Der zweite verwendete Formatstring "HH:MM:SS" zeitigt als Ergebnis eine Uhrzeit im Format Stunden:Minuten:Sekunden, wobei jeweils, falls notwendig, eine 0 vorangestellt wird. Der Doppelpunkt im Formatstring wird bei der Ausgabe wiederum umgesetzt in ein landesspezifisches Zeichen. Auf einem deutschen Windows ist dies allerdings normalerweise ebenfalls ein Doppelpunkt.

Der dritte Formatstring "ww" gibt die laufende Kalenderwoche aus. Da eine nackte Zahl nicht sehr aussagekräftig ist, wird dieser Zahl noch das Wort *Kalenderwoche* vorangestellt.

Ereignisroutine Timer

langes Datumsformat

Uhrzeit

Kalenderwoche

Die Prozedur wird durch den Timer jetzt zyklisch, alle 1000 Millisekunden einmal aufgerufen. Kurze Zeit nach dem Programmstart sehen Sie die korrekten Daten wie in Abbildung 5.62. Da der Timer jede Sekunde einmal abläuft, wird die Ereignisprozedur sekundlich aufgerufen und die Uhr läuft tatsächlich.

Abbildung 5.62:
Die Uhr läuft



5.4 Rechenfunktionen

Wenn Sie Rechnungen durchführen müssen, die über die bereits vorgestellten Grundrechenarten hinausgehen, so lässt Sie Visual Basic nicht allein. Auch hierfür sind die wichtigsten Rechenfunktionen bereits definiert.

Die wichtigsten Rechenfunktionen werden in Tabelle 5.9 anhand des Funktionsnamens und einer textlichen Beschreibung in aller Kürze vorgestellt.

Tabelle 5.8:
Rechenfunktionen

Funktion	Beschreibung
Abs	Berechnet den Absolutwert einer Zahl, also den Zahlenwert ohne Vorzeichen.
Atn	Berechnet den Arcustangens eines Längenverhältnisses.
Cos	Berechnet den Cosinus.
Exp	Berechnet den Exponenten zur Basis e (2,718282).
Fix	Schneidet den Nachkommateil einer Zahl ab. Bei negativen Zahlen wird aufgerundet, d. h. aus -5,6 wird -5.
Int	Schneidet den Nachkommateil einer Zahl ab, rundet bei negativen Zahlen jedoch ab (aus -5,6 wird -6).
Log	Berechnet den natürlichen Logarithmus einer Zahl bezogen auf die Basis e.
Rnd	Ermittelt eine Zufallszahl zwischen 0 und 1.
Sgn	Ermittelt den Vorzeichentyp. Das Ergebnis ist -1 bei negativen, 1 bei positiven Zahlen und 0, wenn auch das Argument 0 ist.
Sin	Berechnet den Sinus.
Sqr	Berechnet die Quadratwurzel einer positiven Zahl.
Tan	Berechnet den Tangens.

5.4.1 Übung: Lottozahlen Quicktip

Ermitteln Sie einen Quicktip für Lotto und geben Sie diesen aus.

Lesen Sie die Visual Basic Online-Hilfe der Funktion *Rnd*, bevor Sie mit der Übung beginnen.



Lösung

Ein Lotto-Quicktip sollte zufällig sein, d.h. wenn Sie die Funktion aufrufen, sollten unterschiedliche Zahlenkombinationen das Ergebnis sein. Hierzu wird die Funktion *Rnd* verwendet.

Rnd ermittelt allerdings immer die gleiche Zahlenreihe, wenn nicht zusätzlich die Funktion *Randomize* vor dem Aufruf von *Rnd* verwendet wird. *Randomize* legt den Startwert für den *Rnd*-Aufruf anhand der Systemuhrzeit fest, und da diese sich normalerweise sehr schnell ändert, ergeben sich immer unterschiedliche Zahlenreihen.

Ein weiteres Problem der *Rnd*-Funktion ist, dass sie Werte zwischen 0 und kleiner 1 liefert. Wir benötigen für einen Quicktip allerdings Zahlenwerte zwischen 1 und 49. Die notwendige Umrechnung errechnet sich laut Visual Basic Online-Hilfe aus folgender Formel:

$$\text{Int}((\text{Obergrenze} - \text{Untergrenze} + 1) * \text{Rnd} + \text{Untergrenze})$$

Umgesetzt auf unseren Fall:

$$\text{Int}((49 - 1 + 1) * \text{Rnd} + 1)$$

Nachdem wir geklärt haben, wie die Zufallszahlen generiert werden, kümmern wir uns um die Programmoberfläche. Wir benötigen eine Schaltfläche, um die Funktion aufzurufen, und eine geeignete Ausgabemöglichkeit. Ich habe mich für eine Liste entschieden, weil diese über die Eigenschaft *Sorted* (Abbildung 5.63) in der Lage ist die einzelnen Einträge sortiert darzustellen.

Randomize setzt »zufälligen« Startwert für Rnd

Umrechnung in gewünschte Werte

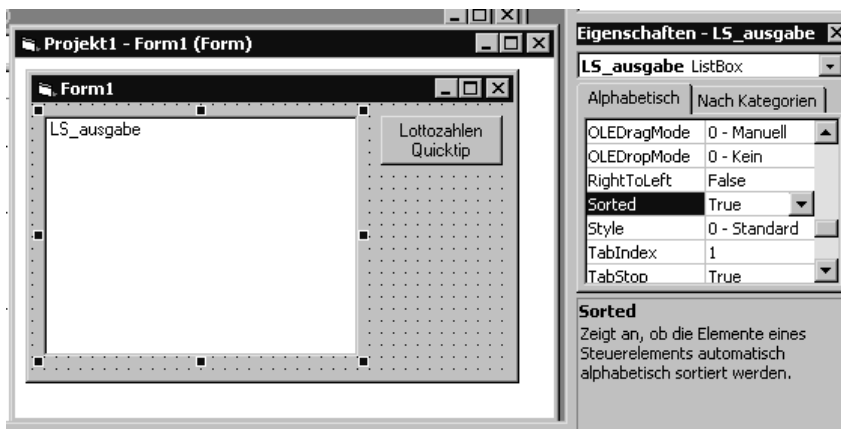


Abbildung 5.63: Programmoberfläche Übung Lotto Quicktip und Eigenschaft *Sorted*



Damit die Sortierung in der Liste für die eingetragenen Zahlen richtig funktioniert, wird der bereits bekannte Trick verwendet, die Zahlen mit vorlaufenden Nullen zu formatieren.

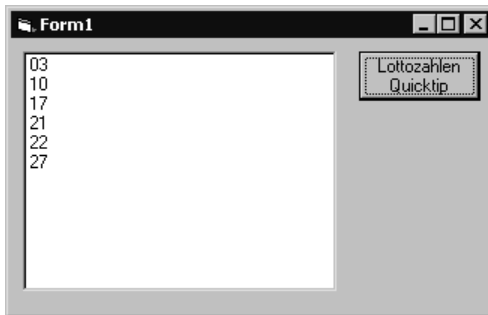
Somit sind alle Vorarbeiten erledigt, und die Ereignisprozedur kann wie folgt programmiert werden:

```
Private Sub BT_Lotto_Click()  
Dim i As Integer  
LS_ausgabe.Clear  
Randomize  
For i = 1 To 6  
    LS_ausgabe.AddItem Format(Int(49 * Rnd + 1), "00")  
Next i  
End Sub
```

Zu dem bereits Besprochenen ist lediglich eine *For...Next*-Schleife hinzugekommen, die bei einem Aufruf gleich die benötigten 6 Zahlen ermittelt.

Die Ausgabe eines Quicktipps sehen Sie in Abbildung 5.64.

Abbildung 5.64:
Ausgabe eines
Lotto Quicktipps



Das oben beschriebene Programm ist nicht fehlerfrei. Die *Rnd*-Funktion liefert eventuell auch gleiche Zahlen innerhalb einer Reihe. Bei einem Lotto Quicktipp kann aber nicht mehrmals die gleiche Zahl angekreuzt werden, so dass in diesem Fall eine Zahl zu wenig ermittelt wird.

Im Grunde müsste das Programm eine doppelte Zahl erkennen und so lange Zufallszahlen generieren, bis sechs unterschiedliche Zahlen zwischen 1 und 49 ermittelt wurden.

Die Wahrscheinlichkeit, dass der Fehler passiert, ist nicht klein (6 zu 49). Wenn Sie also mögen, können Sie das Programm noch verbessern.

5.5 Umwandlungsfunktionen

Wenn Variablen unterschiedlichen Datentyps einander zugewiesen werden, müssen diese vorher umgewandelt werden. Visual Basic macht dies sehr oft automatisch.

Trotzdem kennt Visual Basic eine erstaunlich große Anzahl an Umwandlungsfunktionen. In Tabelle 5.10 werden die wichtigsten kurz vorgestellt.

Funktion	Beschreibung
Asc	Ermittelt den ASCII-Code des ersten Zeichens eines Strings.
Cbool	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Boolean</i> um.
Cbyte	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Byte</i> um.
Ccur	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Currency</i> um.
Cdate	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Date</i> um.
Cdec	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Decimal</i> um.
Cdbl	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Double</i> um.
Cerr	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Error</i> um.
Cint	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Integer</i> um.
CLng	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Long</i> um.
CSng	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Single</i> um.
CStr	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>String</i> um.
Cvar	Wandelt einen numerischen Ausdruck in eine Zahl des Datentyps <i>Variant</i> um.
Hex	Wandelt einen numerischen Ausdruck in eine Zeichenkette, die eine hexadezimale Darstellung des numerischen Ausdrucks beinhaltet, um.
Oct	Wandelt einen numerischen Ausdruck in eine Zeichenkette, die eine oktale Darstellung des numerischen Ausdrucks beinhaltet, um.

Tabelle 5.9:
Umwandlungsfunktionen

5.6 Programmieren von Prozeduren und Funktionen

In den bisherigen Programmbeispielen und Übungen wurden immer nur wenige Zeilen Programmcode geschrieben. Das ganze Programm oder die Teilübung wurde immer in *einer* Prozedur komplett ausgeführt.

Ein größeres Projekt kann aber sehr leicht mehrere tausend Zeilen Programmcode beinhalten. Wenn Sie diese in *eine* Prozedur schreiben würden, wären Sie sehr lange damit beschäftigt, etwas zu suchen.

Programme strukturieren

Daher gibt es in Visual Basic die Möglichkeit ein Programm zu strukturieren. Es werden große Funktionsblöcke in kleine logische Einheiten aufgeteilt. Diese kleinen Einheiten nennt man ebenfalls Funktionen (oder Prozeduren). Allerdings sind diese nicht von Visual Basic vorgegeben, sondern werden von Ihnen selbst geschrieben.

Sourcecode reduzieren

Durch das Schreiben von Funktionen ergibt sich ein weiterer Vorteil. Wenn eine bestimmte Aufgabe mehrfach an verschiedenen Programmstellen ausgeführt werden muss, kann diese Aufgabe in eine Funktion ausgelagert werden. Diese kann dann so oft wie nötig an den entsprechenden Stellen aufgerufen werden. Damit haben Sie nicht nur einen strukturierten, sondern auch einen reduzierten Sourcecode.

Unterschied Prozedur und Funktion

Es gibt in Visual Basic zwei grundsätzliche Ausprägungen von Funktionen. Die *Prozedur* und die *Funktion*. Ein *Prozedur* unterscheidet sich von einer *Funktion* lediglich dadurch, dass die *Funktion* ein Ergebnis liefert, während die *Prozedur* dies nicht macht.

Da Funktionen ein Ergebnis liefern, können sie im Programmcode an allen Stellen eingesetzt werden, an denen auch eine Variable des Ergebnis-Datentyps eingesetzt werden könnte.

Drei wesentliche Vorteile sprechen für die Verwendung von *Funktionen* und *Prozeduren*:

- ▶ Durch sie werden Programme in kleine logische Einheiten aufgeteilt. Diese können, jede für sich, einfacher getestet und mehrfach eingesetzt werden.
- ▶ Durch die Reduzierung der Sourcen und die Verwendung sprechender Prozedurnamen steigt die Übersichtlichkeit von Programmen. Die Fehlersuche reduziert sich auf kleine Einheiten.
- ▶ Allgemeine Funktionen und Prozeduren können ohne oder mit nur wenigen Anpassungen in anderen Programmen verwendet werden. Sie werden hierzu lediglich in einem eigenen Modul gespeichert und in einem neuen Programm aus diesem Modul geladen.

5.6.1 Prozeduren

Verwenden Sie Prozeduren, wenn Ihre Anweisungen kein Ergebnis haben.

Auch wenn Sie nicht annehmen, dass Sie eine bestimmte Aufgabe mehrfach in Ihrem Programm benötigen, kann es sinnvoll sein, einen zusammengehörigen Teil von Anweisungen in eine Prozedur auszulagern. Die Übersichtlichkeit Ihres Programms wird auf jeden Fall verbessert.



```
[Private | Public [Static] Sub Prozedurname [(Parameterliste)]
    [Anweisungen]
    [Exit Sub]
    [Anweisungen]
End Sub
```

Syntax

Eine Prozedur ist eine Sammlung von Anweisungen, die durch eine Definitionszeile (enthält das Schlüsselwort *Sub* und den Prozedurnamen) und die Schlüsselwörter *End Sub* eingekleidet sind. Wird die Prozedur aufgerufen, werden alle Anweisungen zwischen *Sub* und *End Sub* ausgeführt. Ein vorzeitiger Abbruch der Funktion kann über die Schlüsselwörter *Exit Sub* auch innerhalb der Prozedur erfolgen.

Die Definitionszeile enthält zudem ein Schlüsselwort für den Gültigkeitsbereich der Prozedur und eine Aufrufparameterliste. Durch den Gültigkeitsbereich wird festgelegt, wer, oder besser von wo aus die Prozedur verwendet werden kann.

Die Aufrufparameterliste wird verwendet um Variablen und Werte zu übergeben. Dieses Vorgehen wurde bereits bei den internen Funktionen von Visual Basic praktiziert. Die Parameterübergabe bei selbst geschriebenen Prozeduren unterscheidet sich davon nicht.

Der Gültigkeitsbereich von Prozeduren

Sie können Prozeduren in Standard-, in Klassen- und in Formular-Modulen platzieren. Der Standardwert des Gültigkeitsbereichs ist *Public*, d.h. öffentlich. Sie können somit von jeder anderen Stelle im Programm aus aufgerufen werden.

Ist die öffentliche Prozedur in einem Standardmodul hinterlegt, so kann sie ohne weitere Angabe direkt aufgerufen werden. Daraus abgeleitet ergibt sich die Notwendigkeit einen projektweit eindeutigen Namen für die Prozedur zu definieren.

Ist die Prozedur in einem anderen Modul des Projekts untergebracht, so muss der Aufruf ergänzt werden durch einen vorangestellten Modulnamen. Vergessen Sie den Modulnamen, findet Visual Basic die Prozedur nicht. Bei Prozeduren innerhalb von Formmodulen ist es möglich, gleiche Namen zu vergeben.

Folgende Programmzeilen zeigen den Unterschied zwischen Public-Funktionen in einem Standardmodul und in einem anderen Modul des Projekts:

```
playSignal()  
Form2.playSignalForm2()
```

Die Prozeduraufrufe erfolgen im oberen Beispiel in einem anderen Modul des Projekts. Durch den ersten Aufruf kann nur eine Prozedur aufgerufen werden, die in einem Standardmodul hinterlegt ist. Der zweite Aufruf adressiert eine Prozedur in einem *Formular* mit Namen *Form2*.

Wird die Prozedur *playSignalForm2()* innerhalb des Formulars *Form2* verwendet, so kann der Objektname entfallen. Der Aufruf würde also wie folgt aussehen:

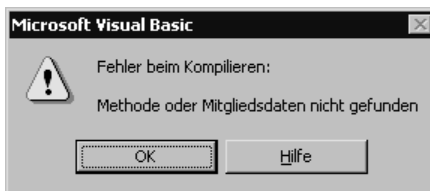
```
playSignalForm2()
```

Es ist durchaus möglich, dass Sie in einem Formular auch lokale Prozeduren schreiben, die nicht von anderen Modulen aus aufrufbar sein sollen. In diesem Fall verwenden Sie bei der Deklaration der Prozedur in der Definitionszeile das Schlüsselwort *Private*. Ist eine Prozedur *Private*, so kann sie nur innerhalb des Moduls verwendet werden, in dem sie definiert ist.

Wird beispielsweise die Prozedur *playSignalForm2()* mit dem Schlüsselwort *Private* deklariert, so erhalten Sie bei Aufruf folgender Programmzeile die Fehlermeldung aus Abbildung 5.65.

```
Form2.playSignalForm2()
```

Abbildung 5.65:
Versuch auf eine
private Prozedur
zuzugreifen



Das Schlüsselwort *Static*

Normalerweise werden die lokalen Variablen einer Prozedur bei jedem Aufruf erneut angelegt bzw. initialisiert. Der Speicherbereich dieser Variablen wird nach Beenden der Prozedur freigegeben. Lokale Variablen sind also nur existent, während die Prozedur abgearbeitet wird.

Variablenwerte bleiben erhalten

In bestimmten Situationen kann es jedoch notwendig sein, dass die lokalen Variablen einer Prozedur ihren Inhalt nicht verlieren. Die Variablen haben dann beim nächsten Aufruf der Funktion ihre alten Werte. Um dies zu erreichen, muss das Schlüsselwort *Static* in der Definitionszeile der Prozedur verwendet werden.

Die Parameterliste

Die Parameterliste definiert, welche und wie viele Variablenwerte oder Variablen von der aufrufenden Funktion mitgegeben werden müssen. Die Liste kann

komplett entfallen, wenn die Prozedur keine Parameter benötigt. Es können aber auch nur ein Parameter oder mehrere, durch Komma getrennte Parameter übergeben werden.

Folgende Schlüsselworte können pro Variable in der Liste verwendet werden.

```
[Optional] [ByVal | ByRef] VarName[( )] [As Typ] [= Standardwert]
```

Bei der Definition einer Prozedur wird für jeden Übergabeparameter ein Name (*VarName*) angegeben. Unter diesem Namen kann innerhalb der Prozedur der übergebene Wert bearbeitet werden. Die aufrufende Funktion muss diesen Namen jedoch nicht kennen. Für die aufrufende Funktion ist lediglich der Datentyp des Übergabeparameters von Interesse, da dieser bei der Übergabe beachtet werden muss.

Grundsätzlich können zwei Varianten der Variablenübergabe gewählt werden:

- ▶ Die Übergabe nach Wert, d.h. die aufgerufene Prozedur kopiert den Wert der übergebenen Variablen in eine lokale Variable mit dem Namen des Übergabeparameters. Die Originalvariable wird nicht geändert. Um dies zu erreichen, muss dem Variablennamen das Schlüsselwort *ByVal* vorangestellt werden.
- ▶ Die Übergabe nach Referenz, d.h. die aufrufende Prozedur erhält eine Referenz auf die Originalvariable. Innerhalb der aufgerufenen Prozedur wird die Originalvariable unter dem Namen des Übergabeparameters angesprochen und kann auch verändert werden. Diese Art der Übergabe wird durch das Schlüsselwort *ByRef* erreicht. Da dies die Standardeinstellung von Visual Basic ist, kann dieses Schlüsselwort allerdings auch weggelassen werden.

Das Schlüsselwort *Optional* gibt an, ob ein Parameter übergeben werden muss oder ob er auch ausgelassen werden kann. Innerhalb der Prozedur kann dann über die Funktion *IsMissing* geprüft werden, ob der Parameter übergeben wurde oder nicht.

**optionale
Parameter**

Die Funktion *IsMissing* liefert nur dann ein korrektes Ergebnis, wenn der Übergabeparameter ein Datentyp *Variant* ist. Ansonsten werden die Variablenwerte, wenn sie nicht übergeben wurden, mit Defaults vorbelegt und die Funktion *IsMissing* liefert als Ergebnis, dass die Variable übergeben wurde.



Manchmal genügt es auch, einem fehlenden optionalen Parameter einen festen Defaultwert zuzuweisen. In diesem Fall reicht es in der Definitionszeile der Prozedur dem optionalen Parameter einen Defaultwert zuzuweisen. Hierzu wird dem Datentyp des Parameters ein Gleichheitszeichen nachgestellt und eine Zuweisung mit dem Defaultwert gemacht.

Die Zuweisung eines Defaultwertes funktioniert auch bei anderen Datentypen als dem Variant.



Ein wichtiger Punkt bei der Verwendung von optionalen Parametern ist, dass in der Parameterliste nach dem ersten optionalen Parameter alle weiteren Parameter ebenfalls optional sein müssen.

optionale Parameter ans Ende der Liste

Wenn Sie also einer Prozedur einige nicht optionale und einige optionale Parameter übergeben wollen, so müssen die nicht optionalen Parameter am Anfang der Parameterliste stehen.

Beim Aufruf einer Prozedur können auch komplexe Ausdrücke übergeben werden. D.h. statt einer Konstanten oder Variablen wird eine Berechnung übergeben. Folgende Prozeduraufrufe sind daher identisch:

```
Wurzel(9)
Wurzel(5 + 4)
```

Da der Ausdruck $5 + 4$ vor der Übergabe an die Prozedur von Visual Basic ausgewertet wird, erhält die Prozedur in beiden Fällen als Übergabe den Wert 9.

Aufrufen von Prozeduren

Ist eine Prozedur deklariert, so sollte sie auch verwendet, also aufgerufen werden. Der Aufruf einer Prozedur kann auf mehrere Arten erfolgen:

```
Call Prozedurname [(Argumentenliste)]
Prozedurname [Argumentenliste]
Objektname.Prozedurname [Argumentenliste]
```

Im Grunde sind alle drei Aufrufarten gleichwertig. Die dritte Schreibweise muss immer dann verwendet werden, wenn auf eine Prozedur zugegriffen wird, die sich nicht im eigenen Modul oder einem Standardmodul befindet.



Der Aufruf mit dem Schlüsselwort `Call` hat keine spezielle Funktionsweise und ist eigentlich nur dann sinnvoll, wenn Sie innerhalb Ihres Programm den Aufruf selbst geschriebener Prozeduren durch diese Aufrufart kenntlich machen möchten.



5.6.2 Übung: Prozedur Textumdrehen

Schreiben Sie eine Prozedur, die einen vom Anwender eingegebenen Text umdreht und in einem Dialog ausgibt.

Geben Sie den übergebenen Text in der aufrufenden Prozedur nochmals aus. Verwenden Sie zur Parameterübergabe einmal das Schlüsselwort *ByVal* und einmal das Schlüsselwort *ByRef*. Weisen Sie innerhalb der Prozedur den gedrehten Text dem Übergabeparameter zu.

Lösung

Um die Übung zu lösen, wird ein Texteingabefeld benötigt. Zudem sollen die beiden Prozeduraufrufe über getrennte Schaltflächen aufrufbar sein. Die entsprechende Programmoberfläche sehen Sie in Abbildung 5.66.

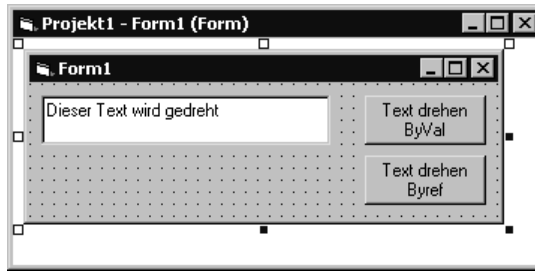


Abbildung 5.66:
 Programmoberfläche der Übung
 Prozedur
 Textumdrehen

Da die Deklaration der Übergabeparameter nicht zur Laufzeit eines Programms gesetzt oder verändert werden kann, werden für die unterschiedlichen Parameterübergaben zwei verschiedene Prozeduren geschrieben.

Die erste Prozedur heißt *textUmdrehenByVal*, ihr wird der Parameter per Wert übergeben:

```
Public Sub textUmdrehenByVal(ByVal text As String)
    Dim i As Integer
    Dim gedrehterText As String
    For i = Len(text) To 1 Step -1
        gedrehterText = gedrehterText + Mid(text, i, 1)
    Next i
    text = gedrehterText
    MsgBox "textUmdrehenByVal" & vbCrLf & gedrehterText
End Sub
```

Aufgerufen wird diese Prozedur in der Ereignisroutine *Click* der Schaltfläche *Textdrehen ByVal*:

```
Private Sub BT_ByVal_Click()
    Dim strHelp As String
    strHelp = TX_Eingabe.text
    textUmdrehenByVal strHelp
    MsgBox "Lokale Variable in aufrufender Prozedur:" & vbCrLf & strHelp
End Sub
```

Der Programmablauf ist folgendermaßen: Nach dem Programmstart wird die Oberfläche aus Abbildung 5.67 angezeigt. In der TextBox ist bereits ein Standardtext eingetragen. Der Anwender kann diesen jetzt durch einen beliebigen anderen Text ersetzen. In den folgenden Abbildungen wird der Programmablauf allerdings mit dem Standardeintrag gezeigt.



Abbildung 5.67:
 Programm wurde
 gestartet,
 Standardtext ist
 eingetragen

Wenn die Schaltfläche *Text drehen ByVal* betätigt wird, wird in der Ereignisprozedur dieser Schaltfläche zunächst der Inhalt des Textes in die lokale Hilfsvariable *strHelp* kopiert. Danach wird die Prozedur *textUmdrehenByVal* aufgerufen, dabei wird die Hilfsvariable *strHelp* übergeben.

Innerhalb der Prozedur *textUmdrehenByVal* wird zunächst der übergebene Text in der bereits früher besprochenen For...Next-Schleife umgedreht und der lokalen Hilfsvariable *gedrehterText* zugewiesen.

Nach getaner Arbeit wird die Variable *gedrehterText* dem Übergabeparameter *text* zugewiesen. Danach erfolgt die Ausgabe des gedrehten Textes über die lokale Hilfsvariable (Abbildung 5.68) innerhalb der Prozedur.

Abbildung 5.68:
Gedrehter Text
innerhalb der
Prozedur



Wird im Dialog aus Abbildung 5.68 die Schaltfläche *Ok* betätigt, geht der Programmablauf nach dem *End Sub* in der Ereignisroutine der Schaltfläche mit der Programmzeile nach dem Prozeduraufruf weiter. Dort wird die lokale Variable *strHelp*, die als Übergabeparameter diente, in einem Dialog ausgegeben.

Abbildung 5.69:
Übergabe-
parameter ist nach
Prozeduraufruf
unverändert bei
ByVal-Übergabe



Wie Sie in Abbildung 5.69 sehen können, wurde die lokale Variable *strHelp* in der Prozedur *textUmdrehenByVal* nicht verändert, obwohl dort eine Zuweisung gemacht wurde. Dies ist ein Ergebnis der Parameterübergabe durch *ByVal*.



Die übergebene Variable wurde in der Prozedur *textUmdrehenByVal* kopiert. Bei der Zuweisung wurde die lokale Kopie des Übergabeparameters verändert, nicht das Original.

Jetzt wird die Schaltfläche *Text drehen ByRef* aufgerufen. Die Unterschiede im Sourcecode sehen Sie in folgenden Zeilen:

```
Public Sub textUmdrehenByRef(ByRef text As String)
...
MsgBox "textUmdrehenByRef" & vbCrLf & gedrehterText
...
End Sub
```



```
Private Sub BT_Byref_Click()
...
    textUmdrehenByRef strHelp
...
End Sub
```

Wie Sie sehen, haben sich lediglich der Prozedurname und das Schlüsselwort bei der Übergabeparameter-Deklaration geändert. Der Programmablauf ist bei dieser Prozedur derselbe wie vorher. Der gedrehte Text wird zunächst in der aufgerufenen Prozedur ausgegeben (Abbildung 5.70).



Abbildung 5.70: Gedrehter Text innerhalb der Prozedur `textUmdrehenByRef`

Danach erfolgt die Ausgabe der Hilfsvariablen in der aufrufenden Prozedur. Wie Sie in Abbildung 5.71 sehen können, wurde diesmal die lokale Hilfsvariable durch die Zuweisung in der aufgerufenen Prozedur verändert. Dies ist eine Folge der Parameterübergabe per Referenz.



Abbildung 5.71: Übergabeparameter ist nach Prozeduraufruf verändert bei `ByRef`-Übergabe

5.6.3 Übung: Positioniere ein Formular

Schreiben Sie eine allgemeingültige Prozedur, die ein Formular (Objekttyp Form) auf dem Bildschirm positioniert.

Wird keine Position übergeben, soll der Dialog horizontal in der Bildschirmmitte platziert werden. Die vertikale Position ist 300 Twips unterhalb der oberen Bildschirmkante.

Schreiben Sie ebenfalls ein Testprogramm für die Funktion.

Lösung

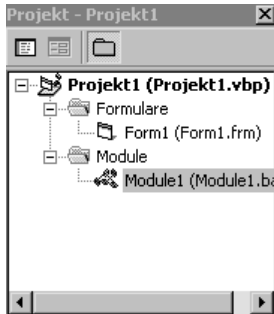
Um die Übung zu lösen, muss nur ein Projekt erstellt werden. Da die Funktion allgemeingültig sein soll, wird sie in einem Standardmodul untergebracht. Für das Testprogramm wird ein normales Formular des gleichen Projektes benutzt.



optionale Parameter

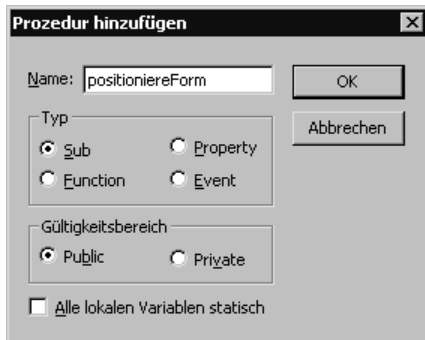
Wenn Sie mit der Programmierung beginnen, sollte Ihr Projekt-Explorer die Einträge haben wie in Abbildung 5.72 zu sehen.

Abbildung 5.72:
Die Projektdateien
der Übung Positioniere ein Formular



Öffnen Sie das Modul im Projekteditor. Danach können Sie die Prozedur komplett selbst schreiben, oder Sie können sich von Visual Basic einen Prozedurrumpf erstellen lassen. Hierzu starten Sie den Dialog PROZEDUR HINZUFÜGEN (Abbildung 5.73) über das Menü EXTRAS->PROZEDUR HINZUFÜGEN...

Abbildung 5.73:
Dialog Prozedur
hinzufügen



Sie können im Dialog PROZEDUR HINZUFÜGEN einen Prozedurnamen eingeben, den Typ der Prozedur und deren Gültigkeitsbereich festlegen. Zudem können Sie einstellen, ob die Variablen der Struktur statisch (Schlüsselwort *Static*) sein sollen.

Wenn Sie den Dialog mit *Ok* bestätigen, erstellt Visual Basic folgenden Prozedurrumpf:

```
Public Sub positioniereForm()  
  
End Sub
```



Zugegeben, in diesem Fall wäre es einfacher gewesen, den Prozedurrumpf selbst zu schreiben, aber bei Ereignissen oder Events lohnt sich der Dialog manchmal.

Jetzt wird der Prozedurrumpf gemäß der Übung ausgefüllt. Zunächst wird die Übergabeparameterliste in der Definitionszeile erstellt.

Um ein Formular zu positionieren, müssen die Eigenschaften *Top* und *Left* gesetzt werden. Wenn aber eine Prozedur außerhalb des Formulars dies tun möchte, so muss diese Zugriff auf das Objekt des Formulars haben.



Jedes Objekt in Visual Basic kann auch als Variable deklariert oder in einer Übergabeparameterliste verwendet werden. Sie müssen lediglich wissen, welchen Datentyp das Objekt hat. Ein einfacher Weg, der immer funktioniert, wenn das Objekt eine Oberfläche hat, ist das Eigenschaftfenster. Sie aktivieren das betreffende Objekt im Designer und sehen nach, was im Eigenschaftfenster neben dem Objektamen steht. Dieser Text kann als Datentyp in einer Deklaration verwendet werden.



Abbildung 5.74 zeigt das Eigenschaftfenster für das Objekt Form1 unseres Programms. Wie Sie sehen, hat das Objekt den Objekttyp *Form*. Die Deklaration muss also heißen:

```
Dim einFormular as Form
```

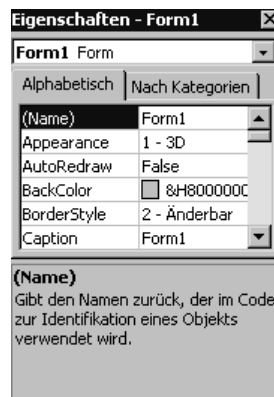


Abbildung 5.74: Objekttyp ermitteln über Eigenschaftfenster

Zwei weitere Übergabeparameter sind notwendig, um die Form zu positionieren. Die Position von *Top* und die Position von *Left*. Da diese beiden Parameter optional sein sollen, wird bei beiden das Schlüsselwort *Optional* verwendet.

Sind die beiden Werte nicht gesetzt, so soll die horizontale Ausrichtung mittig zum Bildschirm sein. Da wir die Bildschirmgröße nicht kennen, muss diese zur Laufzeit des Programms ermittelt werden. Daher wird für den Parameter *x* ein *Variant* als Datentyp gewählt, damit die Prüfung über die Funktion *IsMissing* funktioniert.



Wenn der Übergabeparameter für die vertikale Richtung nicht übergeben wurde, so muss nichts berechnet, sondern ein fixer Wert von 300 gesetzt werden. In diesem Fall genügt eine Defaultwert-Zuweisung in der Parameterliste.

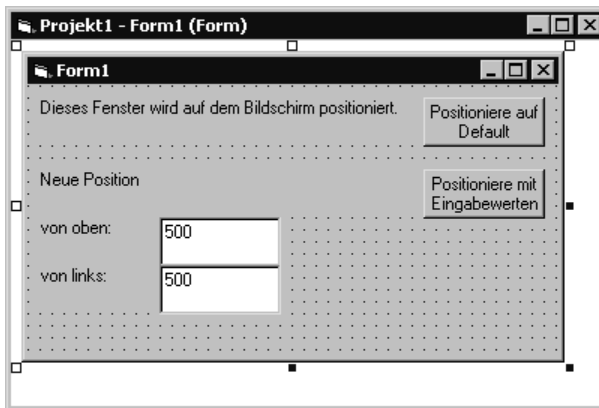
Die komplette Prozedur sehen Sie in folgenden Programmzeilen:

```
Public Sub positioniereForm(formx As Form, Optional x, Optional y As Integer = 300)
    If IsMissing(x) Then
        formx.Left = (Screen.Width - formx.Width) / 2
    Else
        formx.Left = x
    End If
    formx.Top = y
End Sub
```

Die Berechnung der mittigen Position erfolgt über das Screen-Objekt. Dies liefert die Breite des Bildschirms zur Laufzeit, so dass das Programm bei jeder Bildschirmauflösung funktioniert.

Um das Programm zu testen, wird ein Formular benötigt, welches zumindest zwei Schaltflächen hat. Eine Schaltfläche wird verwendet, um den Dialog in seinen Defaultwerten zu positionieren. Die andere wird verwendet, um beim Funktionsaufruf die Positionswerte zu übergeben. Die Programmoberfläche aus Abbildung 5.75 lässt zudem die Eingabe der Positionswerte zu.

Abbildung 5.75:
Programmoberfläche Übung
Positioniere ein
Formular



In der Ereignisprozedur der Schaltfläche *Positioniere auf Default* muss lediglich der Prozeduraufruf erfolgen. Sie sehen die Prozedur in den folgenden Zeilen:

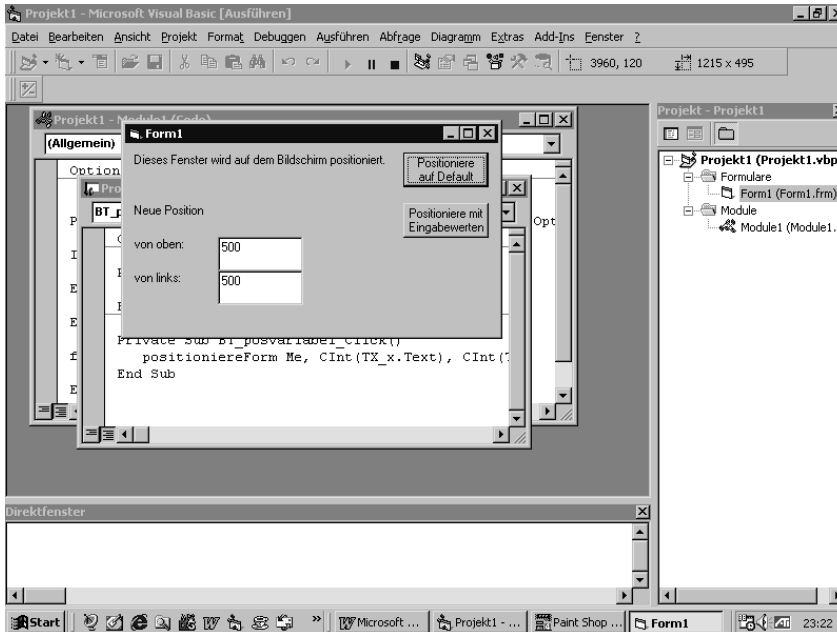
```
Private Sub BT_posfix_Click()
    positioniereForm Me
End Sub
```

Beim Funktionsaufruf wurden die Parameter *x* und *y* weggelassen. Die Prozedur muss also für beide Werte seine Defaults einsetzen.

Für den Parameter Form wird das Schlüsselwort *Me* verwendet. Dahinter verbirgt sich eine Referenz auf das eigene, momentan aktive Objekt.



Die Auswirkung dieses Aufrufs können Sie in Abbildung 5.76 und Abbildung 5.77 sehen.



*Abbildung 5.76:
Position des
Formulars vor Auf-
ruf der Default-
positionierung*

Abbildung 5.76 zeigt das Formular und seine Position auf dem Bildschirm nach dem Start des Programms.

In Abbildung 5.77 wurde das Formular horizontal mittig ausgerichtet. Zudem wurde die Position auf den korrekten Wert in vertikaler Richtung gesetzt.

Im Folgenden sehen Sie die Ereignisprozedur der Schaltfläche Positioniere mit Eingabewerten. Auch hier handelt es sich wieder lediglich um den Aufruf der Funktion.

```
Private Sub BT_posvariabel_Click()
    positioniereForm Me, CInt(TX_x.Text), CInt(TX_y.Text)
End Sub
```

Es wurden allerdings diesmal auch die Parameter x und y übergeben. Diese werden aus den beiden *TextBoxen* der Oberfläche ermittelt und vor der Übergabe in den Datentyp *Integer* konvertiert.

Das Ergebnis des Funktionsaufrufs mit den Defaultwerten 500/500 sehen Sie in Abbildung 5.78.

5 Workshop: Prozeduren und Funktionen

Abbildung 5.77:
Position des
Formulars nach
Aufruf der Default-
positionierung

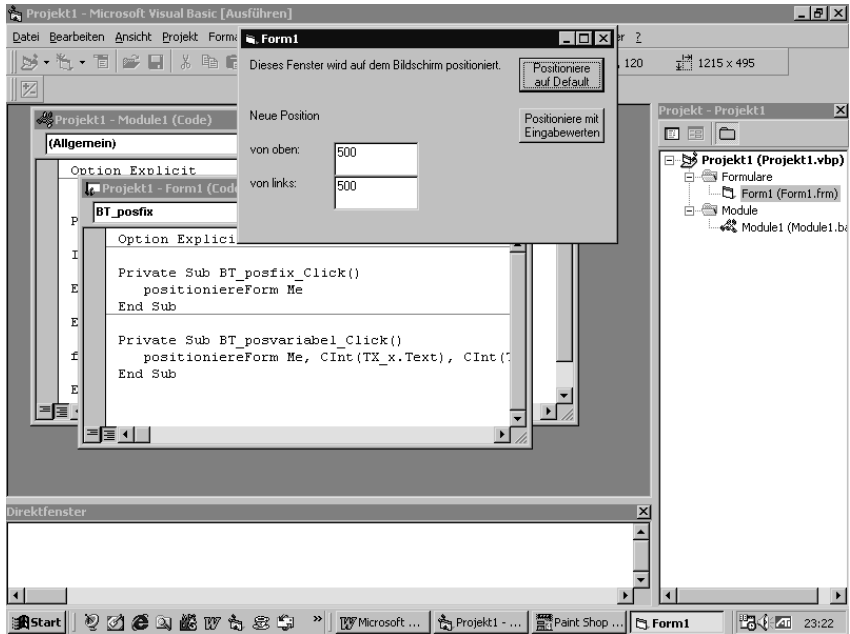
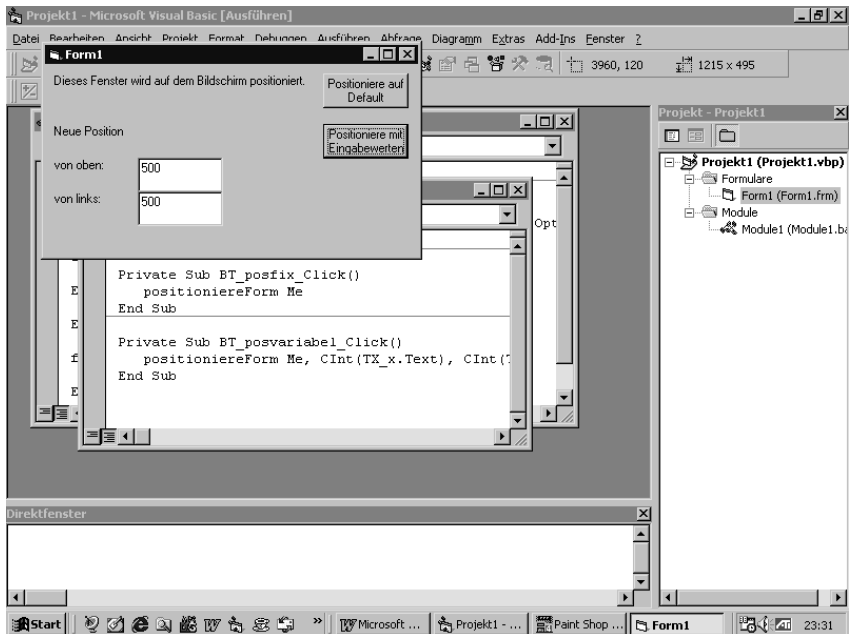


Abbildung 5.78:
Position des Formu-
lars nach Aufruf mit
Werten 500/500



Wie Sie sehen, hat auch bei diesem Aufruf die Prozedur das getan, was von ihr erwartet wurde.

5.6.4 Funktionen

Zwischen einer Funktion und einer Prozedur gibt es nur einen entscheidenden Unterschied. Eine Funktion liefert ein Ergebnis, eine Prozedur nicht.

Funktion hat Rückgabewert bzw. Ergebnis

Eine Funktion wird normalerweise in einer expliziten oder impliziten Zuweisung aufgerufen, damit das Ergebnis vom Aufrufenden auch benutzt werden kann.

```
[Public | Private ] [Static] Function Funktionsname
[(Argumentenliste)] [As Type]
    [Anweisungen]
    [Funktionsname = Ausdruck]
    [Anweisungen]
    [Exit Function]
    [Anweisungen]
    [Funktionsname = Ausdruck]]
End Function
```

Syntax

Die Struktur einer Funktion ist nahezu identisch zur Struktur einer Prozedur. Die Schlüsselwörter *Privat*, *Public* und *Static* haben bei einer Funktion die gleiche Bedeutung wie bei einer Prozedur. Auch die Argumentenliste ist gleich wie bei einer Prozedur. Der Unterschied reduziert sich auf die Verwendung des Schlüsselworts *Function* (anstatt *Sub*) und den Zusatz *As Type* am Ende der Definitionszeile.

Wird der Datentyp einer Funktion nicht festgelegt, so nimmt Visual Basic standardmäßig an, dass die Funktion eine Variable des Datentyps *VARIANT* zurückliefert.

Zwei weitere Schlüsselwörter, die sich nur namentlich von denen einer Prozedur unterscheiden, sind *Exit Function* und *End Function*. Mit *Exit Function* wird eine Funktion vorzeitig beendet. Mit *End Function* wird die Funktion normal beendet und sie dient gleichzeitig im Programmcode als Abschluss der Funktion.

Wie aber gibt eine Funktion einen Wert zurück?

Innerhalb einer Funktion wird der Funktionsname wie eine Variable verwendet. Der Wert, der dieser Variablen zugewiesen wird, bevor eine *Exit Function*- oder *End Function*-Anweisung im Programmcode angetroffen wird, wird als Funktionsergebnis an die aufrufende Stelle zurückgeliefert.

Rückgabewert in Funktionsname speichern

Aufrufen von Funktionen

Um eine Funktion aufzurufen, wird ihr Name, gefolgt von der Übergabeparameterliste in Klammern, geschrieben. Dabei unterscheiden sich selbst geschriebene Funktionen nicht von den internen Visual Basic-Funktionen.

Eine Funktion kann in einer expliziten Zuweisung oder in einer impliziten Zuweisung verwendet werden. Vereinfacht könnte man formulieren, dass ein Funktionsaufruf überall da im Programmcode stehen darf, wo eine Variable ihres Rückgabedatentyps stehen könnte.

Folgendes Beispiel ist ein Funktionsaufruf in expliziter Zuweisung:



```
Ergebnis = BerechneFläche(2,4)
```

BerechneFläche ist der Funktionsname, 2 und 4 sind Übergabeparameter der Funktion. Die Variable *Ergebnis* muss den gleichen Datentyp haben wie die Funktion.

Folgende Zeile ist ein Beispiel für einen impliziten Aufruf:

```
MsgBox "Die Fläche ist " & BerechneFläche(2,4)
```

Die Funktion wird hier innerhalb einer String-Verkettung aufgerufen. Das Ergebnis wird nicht in einer Variablen gespeichert, sondern direkt verkettet und dann über die Funktion *MsgBox* ausgegeben.

Da die Funktion sicher einen numerischen Wert zurückliefert, funktioniert obige Programmzeile nur, weil Visual Basic bei der String-Verkettung eine automatische Konvertierung des numerischen Ergebnisses in eine Zeichenkette vornimmt.



5.6.5 Übung: Funktion Textumdrehen

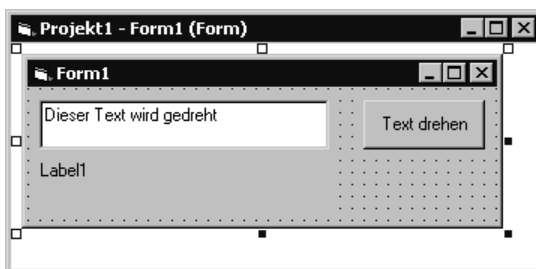
Schreiben Sie eine Funktion, die einen vom Anwender eingegebenen Text umdreht und als Ergebnis zurückliefert.

Lösung

Die eigentliche Übung »Textumdrehen« haben wir schon mehrfach gelöst. Es geht bei dieser Übung also hauptsächlich darum, die Unterschiede zwischen einer Prozedur und einer Funktion herauszuarbeiten.

Für die Oberfläche benötigen wir eine Eingabemöglichkeit für den Anwender und eine Schaltfläche, um die Funktion aufzurufen. Zudem habe ich in Abbildung 5.79 ein Label verwendet um das Ergebnis der Funktion darzustellen.

Abbildung 5.79:
Programmier-
fläche der
Übung Funktion
Textumdrehen



Der Programmcode der Funktion sieht wie folgt aus:

```
Public Function textUmdrehen(text As String) as String  
Dim i As Integer
```



```
Dim gedrehterText As String
For i = Len(text) To 1 Step -1
    gedrehterText = gedrehterText + Mid(text, i, 1)
Next i

TextUmdrehen = gedrehterText
End Function
```

Es ergeben sich folgende Unterschiede zu einer Prozedur:

- ▶ Anstatt des Schlüsselworts *Sub* wird das Schlüsselwort *Function* verwendet.
- ▶ Die Definitionszeile enthält den Zusatz *As String* und definiert damit den Datentyp der Funktion.
- ▶ Die Zeile *TextUmdrehen = gedrehterText* weist einen Rückgabewert zu, der nach Abschluss der Funktion im aufrufenden Programm zur Verfügung steht.

Der Funktionsaufruf erfolgt in der Ereignisroutine der Schaltfläche *Text drehen*. Folgende Programmzeilen zeigen diese:

```
Private Sub BT_textdrehen_Click()
    LB_Ausgabe.Caption = textUmdrehen(TX_Eingabe.text)
End Sub
```

In der Ereignisroutine wird der Rückgabewert der Funktion direkt der Eigenschaft *Caption* des *Labels* zugewiesen. Das Ergebnis dieser Zuweisung sehen Sie in Abbildung 5.80.



Abbildung 5.80:
Die Funktion hat
den Text gedreht

6

Workshop: Mit den Steuerelementen programmieren

Als Steuerelemente bezeichnet man ganz allgemein Elemente einer Benutzeroberfläche, mit deren Hilfe Dialogmasken oder Formulare schnell erstellt werden können. Steuerelemente sind z.B. *CheckBoxen*, *Schieberegler*, *PushButtons* und Ähnliches. Mit diesen visuellen Steuerelementen kann der Software-Entwickler eine Windows-Bedienoberfläche schnell und einfach gestalten.

Ein Steuerelement muss aber nicht zwangsläufig auf der Oberfläche des fertigen Programms erscheinen, also sichtbar sein. Man denke dabei nur an Steuerelemente z.B. zur Steuerung einer seriellen Schnittstelle, die quasi unsichtbar ihren Dienst verrichten.

Des Weiteren kann ein Steuerelement aber auch aus einer Gruppe von anderen Steuerelementen bestehen. Ein Beispiel dafür ist die Standard-Dateiauswahl von Windows, die ja nicht nur aus einem, sondern aus verschiedenen Dialogen besteht.

In letzter Zeit taucht im Zusammenhang mit Steuerelementen auch der Begriff Komponenten, Componentware oder Komponenten-Software auf. Komponenten, sprich Steuerelemente, kann man sich als selbstständige, hochintegrierte kleine Softwaremodule vorstellen.

Sie sind so geschrieben, dass eine sehr einfache Einbindung in neue Anwendungen möglich ist. Es ist sogar denkbar, dass man eine Anwendung vollständig aus bereits bestehenden Modulen (Komponenten, Steuerelementen) zusammenbaut und nur die ihnen gemeinsame, verbindende Oberfläche schreibt.

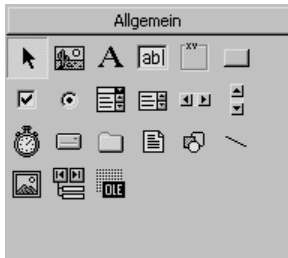
Somit ist es mit Visual Basic 6.0 und den darin enthaltenen Steuerelementen möglich, dass das Entwickeln von Windows-Applikationen mehr einem Montieren von Steuerelementen (Integrieren von Komponenten in einer Benutzeroberfläche) als einem codeschreibenden Programmieren gleichkommt.

6.1 Arbeiten mit der Werkzeugsammlung

Die Werkzeugsammlung beinhaltet alle Steuerelemente in Form von Symbolen. Sie erscheint nach dem Start von Visual Basic links am Bildrand und kann über das Menü ANSICHT->WERKZEUGSAMMLUNG ein- bzw. ausgeblendet werden. Abbildung 6.1 zeigt die Toolbox (Werkzeugsammlung) mit den Standard-Steuerelementen der Learning-, Professional- und Enterprise-Edition.

Mit Hilfe der Werkzeugsammlung können Sie Steuerelemente in Ihre Anwendung einfügen und deren Oberfläche gestalten. Zuerst einmal enthält die Werkzeugsammlung diejenigen Steuerelemente, welche Visual Basic standardmäßig zur Verfügung stellt.

Abbildung 6.1:
Die Werkzeug-
sammlung mit allen
Standard-Steuere-
lementen



Dies sind die so genannten *integrierten Steuerelemente*. Hierzu zählen z.B. das Befehlsschaltflächen-Steuerelement, das Rahmen-Steuerelement, das Textfeld und etliche weitere. Dies sind Steuerelemente, die in der EXE-Datei von Visual Basic enthalten sind. Integrierte Steuerelemente befinden sich grundsätzlich in der Werkzeugsammlung. Sie können auch nicht aus ihr entfernt werden.

Es ist in der Regel vom jeweiligen Projekt abhängig, welche Steuerelemente hinzugefügt werden sollen. Die zusätzlichen Steuerelemente werden als Referenz in der Projektdatei gespeichert, so dass sie beim Laden des Projekts wieder in die Werkzeugsammlung eingefügt werden können.



ActiveX-Steuerelemente sind eigenständige Dateien, die an der Dateinamenerweiterung *.OCX* zu erkennen sind. Die jeweiligen Editionen von Visual Basic enthalten eine Reihe unterschiedlicher ActiveX-Steuerelemente, sprich zusätzlicher Steuerelemente, im Lieferumfang.

Es gibt jedoch auch eine fast unüberschaubar große Menge von Steuerelementen von Drittanbietern, die Sie unabhängig von Ihren Editionen hinzufügen können.



6.1.1 Übung: Zusätzliche Objekte und Steuerelemente einfügen

Ein weiteres Element der Werkzeugsammlung sind *einfügbare Objekte*, so z.B. ein Microsoft Excel Worksheet-Objekt oder ein Microsoft Project Calendar-Objekt.

So soll, wie in Abbildung 6.2. ersichtlich, die Möglichkeit geschaffen werden, das Microsoft Objekt »Chat Room« einer Visual Basic-Applikation zufügen zu können.

Außerdem ist dem Visual Basic-Projekt, wie in Abbildung 6.3. zu sehen, ein weiteres Steuerelement (Microsoft NetShow Player) hinzuzufügen.

Beide Elemente, Objekt »Microsoft Chat Room« und Steuerelement »Microsoft NetShow Player« (Abbildung 6.4), werden unserer Standard-Werkzeugsammlung zugefügt.



Abbildung 6.2:
Das Objekt »Micro-
soft Chat Room«

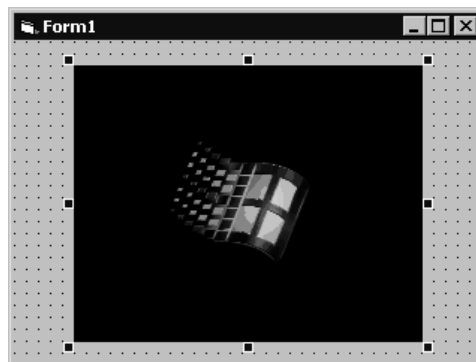


Abbildung 6.3:
Steuerelement
»Microsoft
NetShow Player«



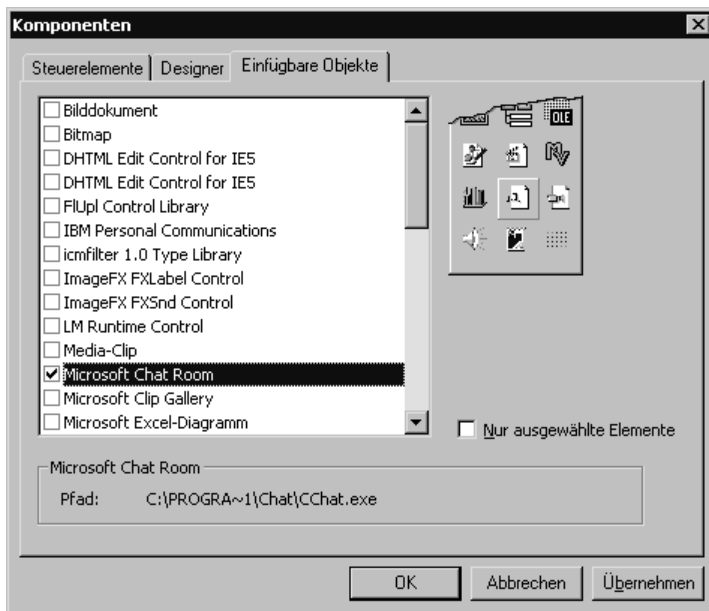
Abbildung 6.4:
Erweiterte Werk-
zeugsammlung

Lösung

Einfügbare Objekte werden durch den Befehl **KOMPONENTEN** aus dem Menü **PROJEKT** angezeigt. Diese Objekte können in die Werkzeugsammlung eingefügt und daher als Steuerelemente betrachtet werden. Etliche solcher Objekte ermöglichen auch die *Automatisierung*.

Sie erlauben Ihnen die Bearbeitung von Objekten anderer Anwendungen aus Ihrer Visual Basic-Anwendung heraus (Abbildung 6.5). Dies ist durch eine recht durchgängige objektorientierte Programmierung der Steuerelemente bezüglich des Betriebssystems möglich.

Abbildung 6.5: Einfügbare Objekte werden durch den Befehl *Komponenten* aus dem Menü *Projekt* angezeigt



Um ein Steuerelement in die Werkzeugsammlung einfügen oder aus ihr entfernen zu können, wird der Komponenten-Dialog aufgerufen. Diesen erreichen Sie über das Menü *PROJEKT->KOMponenten...*

In diesem Fenster werden Ihnen alle verfügbaren Steuerelemente und einfügbaren Objekte angezeigt. Sie sind durch Registerseiten dem Typ nach geordnet. Durch Ankreuzen können die Steuerelemente in die Werkzeugsammlung aufgenommen werden.



Das Fenster erreichen Sie auch mit einem Klick der rechten Maustaste in der Werkzeugsammlung. Im dann erscheinenden Pop-Up-Menü wählen Sie *Komponenten...*

6.2 Mit Steuerelementen im Formular arbeiten

Wenn Sie eine Komponente hinzufügen, so wird ein Symbol innerhalb der Werkzeugsammlung erzeugt. Dies bedeutet jedoch nicht, dass Sie nun sofort auf das Steuerelement und seine Eigenschaften und Funktionen zugreifen können.

Ein Steuerelement in der Werkzeugsammlung ist lediglich das Mittel, um ein Objekt in Ihrer Anwendung zu erzeugen. Hierzu wählen Sie das Symbol des Steuerelements in der Werkzeugsammlung mit einem Doppelklick an.

Daraufhin wird ein Objekt des Steuerelements in der Mitte Ihres aktuellen Formulars erzeugt. Das Objekt befindet sich nun in der Objekthierarchie an oberster Stelle und ist gleichzeitig angewählt. Es kann also sofort hinsichtlich seiner Größe und Position bearbeitet werden.

Hierzu dient das erste Symbol links oben in der Werkzeugsammlung, der Standardzeiger. Dieser wird benötigt, wenn Sie bereits platzierte Steuerelemente anwählen wollen, um diese in Größe und Position zu verändern.

Auch zum Bearbeiten der Eigenschaften eines Objekts muss es angewählt sein. Sie können bei der Platzierung jedoch auch anders vorgehen: Wählen Sie das Symbol in der Werkzeugsammlung mit einem einfachen Klick an und ziehen Sie dann mit gedrückter linker Maustaste einen Rahmen mit den gewünschten Außengrenzen des Objekts auf das Formular. Das neue Objekt füllt dann den gezogenen Rahmen vollständig aus.

Es gibt jedoch auch einige Steuerelemente, deren Größe in einem Formular nicht geändert werden kann. Dies sind in der Regel Elemente, die über keine visuelle Benutzerschnittstelle verfügen.



Das bedeutet, dass diese Elemente, obwohl im Formular als Objekt vorhanden, beim Lauf Ihres Programms nicht sichtbar sind. Beispielhaft sei hier das Timer-Steuerelement genannt.

Sie müssen auf jeden Fall ein Objekt im Formular erzeugen, wenn Sie die Eigenschaften des Steuerelements nutzen wollen, auch wenn es sich dabei um ein *unsichtbares* Steuerelement handelt.

Die in einem Formular (auch Form) erzeugten Objekte können mit der Maus beliebig verschoben und verändert werden. Den Standardzeiger kann man auch dazu benutzen, ganze Gruppen von Objekten einer Form zu bilden.

Ziehen Sie einfach mit gedrückter linker Maustaste einen Rahmen über die Objekte. Alle Objekte, die dabei *berührt* werden, bilden dann beim Loslassen eine Gruppe. Es können nun die gemeinsamen Eigenschaften dieser Gruppe für alle Elemente gleichzeitig im Eigenschaftenfenster bearbeitet werden.



Gruppen sind auch innerhalb der Werkzeugsammlung möglich. Oft kommt es vor, dass bei vielen zum Projekt gehörenden Steuerelementen die Werkzeugsammlung sehr umfangreich und damit unübersichtlich wird. Daher wurde die Möglichkeit geschaffen, die Steuerelemente in Gruppen zu platzieren. Die erste Gruppe heißt *Allgemein* und beinhaltet die grundsätzlichen Steuerelemente der Werkzeugsammlung.





Man erzeugt eine neue Gruppe, indem man mit der rechten Maustaste in die Werkzeugsammlung klickt und im Pop-Up-Fenster *Gruppe hinzufügen* anwählt.

Sie können daraufhin einen Namen für die neue Gruppe vergeben, der dann unten in der Werkzeugsammlung als Balken erscheint. Diese Gruppe umfasst zunächst noch keine Steuerelemente.

Wenn Sie nun ein vorhandenes Steuerelement mit der Maus über diesen Balken ziehen, schaltet die Werkzeugsammlung in die neue Gruppe um und Sie können das Steuerelement dort loslassen.

Die *Allgemein*-Gruppe ist dann nur noch als Balken sichtbar. Ein Anklicken dieses Balkens genügt, und es wird wieder dorthin umgeschaltet. Eine selbst angelegte Gruppe kann auch wieder entfernt werden.

Hierzu klicken Sie mit der rechten Maustaste auf den Balken der Gruppe. Im Pop-Up-Menü können Sie dann *Gruppe entfernen* oder *Gruppe umbenennen* anwählen.

Wenn Sie eine Gruppe löschen, so werden alle darin befindlichen Steuerelemente wieder automatisch in die *Allgemein*-Gruppe eingefügt.



6.2.1 Übung: Richten Sie Steuerelemente aus

Es ist grundsätzlich nicht nur damit getan, die Form mit den aufgabenspezifischen Steuerelementen aus der Werkzeugsammlung zu füllen, sondern es gilt daneben, diese Benutzerschnittstelle auch für den Anwender benutzerfreundlich zu gestalten.

Sehr häufig ist es deshalb notwendig, Steuerelemente oder Gruppen von Steuerelementen an einem bestimmten Objekt auszurichten, damit z.B. alle Steuerelemente untereinander im gleichgroßen Abstand voneinander entfernt sind.

Natürlich können Sie jedes in der Form benutzte Steuerelement einzeln verschieben und somit die Gesamtheit der Objekte ausrichten – aber es geht auch einfacher und vor allem komfortabler.

Starten Sie Visual Basic und wählen Sie als Projekt *Standard-EXE*. Die Überschrift Ihres Formularfensters soll *Ausrichten von Steuerelementen* lauten. Dazu ist die Eigenschaft *Caption* entsprechend zu setzen.

Da dem Anwender zur Auswahl vier Möglichkeiten gegeben werden sollen, integrieren Sie in die Form vier Befehlsschaltflächen. Die jeweiligen Positionen und Objektgrößen sind, wie in Abbildung 6.6 sichtbar, unerheblich, da im Folgenden diese Steuerelement-Gruppe im Formular ohne explizites Verschieben ausgerichtet und in der Größe angepasst werden soll.

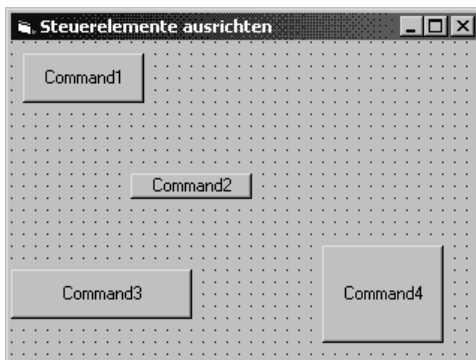


Abbildung 6.6:
Die Form enthält
CommandButtons

Ihre Übung ist es nun, den vier Command-Buttons die gleiche Größe und den pixelgenauen Abstand voneinander zu geben. Außerdem sollen die Befehls-schaltflächen am rechten Rand der Form genau untereinander ausgerichtet werden. Zum Schluss sind die in der Form integrierten Steuerelemente gegen unbeabsichtigtes Verschieben oder unbeabsichtigte Größenanpassungen zu schützen.

Lösung

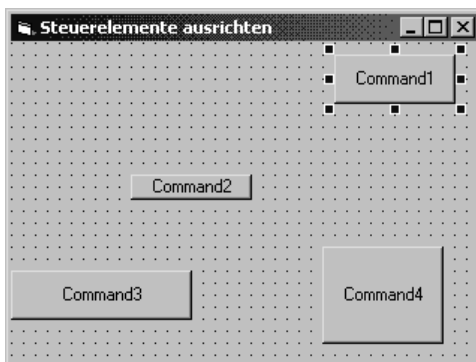


Abbildung 6.7:
Befehlsschaltfläche
1 wird wie
gewünscht
positioniert

Der erste CommandButton wird, weil es unsere Übung erfordert, oben rechts im Formular neu positioniert. Er ist so etwas wie ein »Master«, an dem sich die anderen drei Command-Buttons nachher auszurichten haben. Er muss deshalb auch markiert werden (Abbildung 6.7).

Jetzt gilt es, über die vier Objekte mit gedrückter linker Maustaste zu fahren, um diese als Gruppe zu markieren. Das Steuerelement, an dem ausgerichtet werden soll (CommandButton 1), erhält zum Schluss den aktiven Markierungsrahmen (Abbildung 6.8).

Abbildung 6.8:
Die Steuer-
elemente als
Gruppe markiert

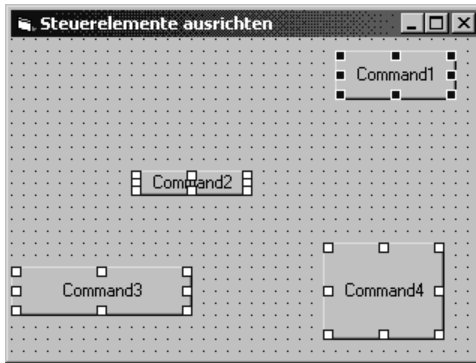
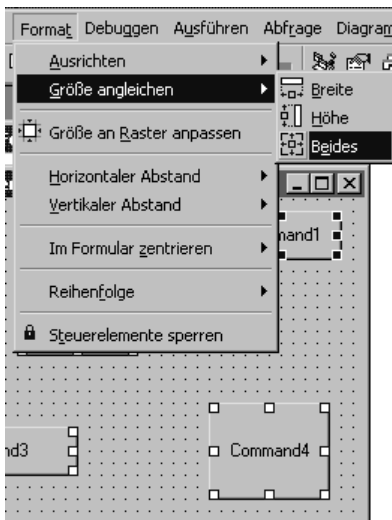


Abbildung 6.9:
Menü-Befehl
Größe angleichen
aus dem Menü
Format



Aus den vorhergehenden Abbildungen ersehen Sie, dass die Befehlsschaltflächen mit unterschiedlicher Größe in das Formular gezeichnet wurden. Damit alle vier Buttons im ersten Schritt die gleiche Größe erhalten, wählen Sie aus dem Menü **FORMAT** den Befehl **GRÖSSE ANGLEICHEN** (Abbildung 6.9).

Optional kann hier entweder nur die Breite, die Höhe oder, wie wir wollen, auch beides angeglichen werden (Abbildung 6.9).

Nach Ausführung der Aktion haben unsere vier Schaltflächen die gleiche Größe und sind immer noch als Gruppe markiert (Abbildung 6.10).

Jetzt sollen die Command-Buttons am Master-Button, der Befehls-Schaltfläche 1, ausgerichtet werden. Dazu ist wieder das Menü **FORMAT** zu aktivieren. Da alle Objekte rechts ausgerichtet werden sollen, wählen Sie den Menü-Befehl **AUSRICHTEN**, Option **RECHTS** (Abbildung 6.11).

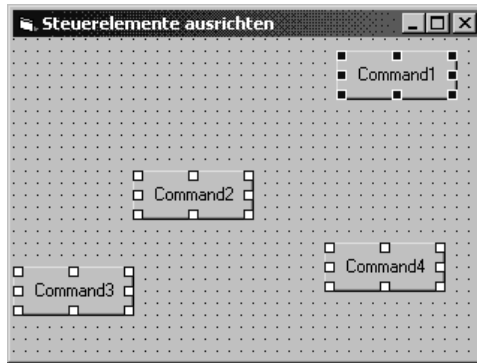


Abbildung 6.10:
Die Objekte sind
immer noch
markiert

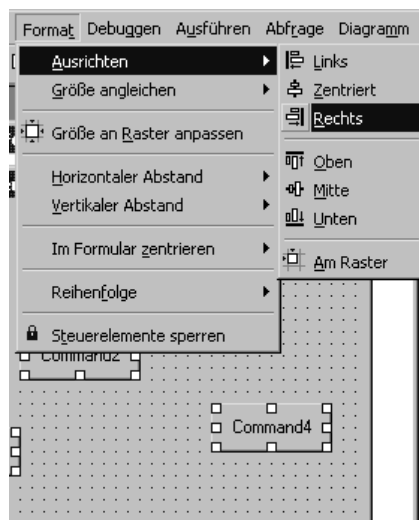


Abbildung 6.11:
Objekte ausrichten

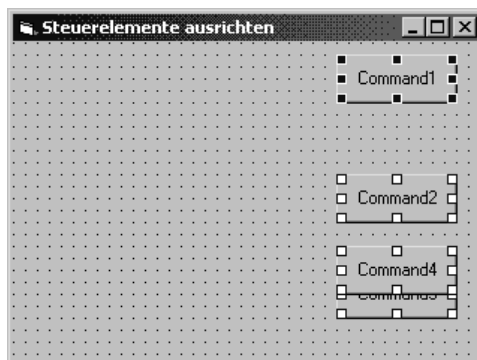
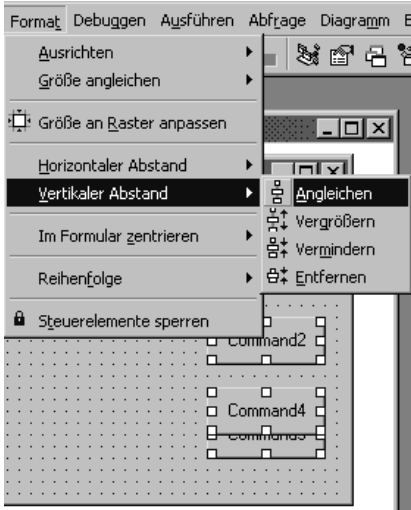


Abbildung 6.12:
Rechts ausgerichtet

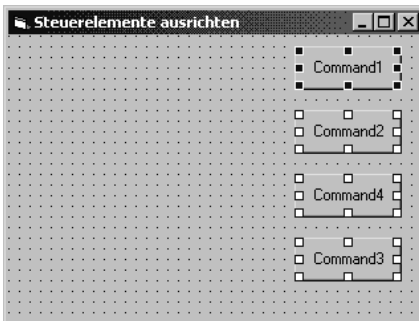
Die Command-Buttons sind nun, wie in Abbildung 6.12 zu sehen, am rechten Rand der Form genau untereinander ausgerichtet und immer noch für weitere Aktionen markiert. Nur der Abstand zwischen den Objekten lässt noch Wünsche offen.

Abbildung 6.13:
Abstand
korrigieren



Diese Unschönheit korrigieren Sie auf einen Schlag mit dem Befehl VERTIKALER ABSTAND, und zwar wieder aus dem Menü FORMAT. Da der Objektabstand überall gleich sein soll, ist die Option ANGLEICHEN zu wählen (Abbildung 6.13).

Abbildung 6.14:
Ausgerichtete
Steuerelemente



In der Abbildung 6.14 sehen Sie nun das Endergebnis. Die vier in der Form verwendeten Objekte haben die gleiche Größe und den pixelgenauen Abstand voneinander.

Nachdem die Oberfläche gestaltet ist und ein benutzerfreundliches Aussehen erhalten hat, können Sie zum Schluss, die Steuerelemente Ihrer Form gegen unbeabsichtigtes Verschieben oder unbeabsichtigte Größenanpassungen schützen, indem Sie, wie in Abbildung 6.15 zu ersehen, den Menü-Befehl STEUERELEMENTE SPERREN aus dem Menü FORMAT aktivieren.

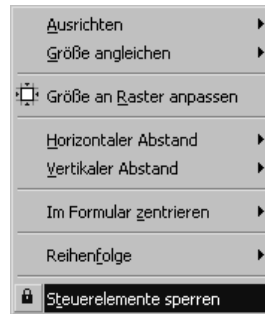


Abbildung 6.15:
Steuerelemente sperren

6.3 Ereignisgesteuert programmieren

Jedes Steuerelement, das Sie in Ihre Anwendung einfügen, entspricht einer Klasse mit gekapselten Daten und einer klar definierten Zugriffsschnittstelle. Diese Schnittstelle besteht neben *Eigenschaften* und *Methoden* zusätzlich aus *Ereignissen*.

Ereignisse sind der einzige Weg für Visual Basic-Objekte (Steuerelemente), selbstständig mit der Außenwelt zu kommunizieren. Doch gerade die Ereignisse sind es, die das Programmieren mit Visual Basic so reizvoll und einfach machen.

Da Microsoft Windows hauptsächlich auf Ereignisse aufgebaut ist, bietet eine Programmierumgebung, die das Verarbeiten von Ereignissen beinhaltet, alle Voraussetzungen für eine einfache und doch effiziente Programmentwicklung.

Dies ist mit ein Grund für den großen Erfolg von Visual Basic. In anderen Programmiersprachen ist die Ereignisverwaltung Sache des Programmierers. Dabei ist dies eine äußerst komplexe Aufgabe und erfordert detaillierte Kenntnisse des Nachrichtensystems von Windows. Ereignisse sind der Teil in Ihrem Programmcode, der das Programm überhaupt am Laufen hält.

Ein Fehlen der Ereignisse würde ein statisches Programmgebilde zur Folge haben, das zwar funktioniert, praktisch jedoch auf keine Aktion seitens des Benutzers reagieren kann. Doch gerade das ist der Sinn eines Programms mit grafischer Benutzeroberfläche, nämlich dem Benutzer die Möglichkeit zur Interaktion zu geben.

Wenn Sie beispielsweise vor einem Programm sitzen, das Sie in einem Fenster zur Eingabe von Daten oder einer Wahl zwischen verschiedenen Optionen auffordert, so wird das Programm so lange nichts tun, bis Sie aktiv werden.

Es befindet sich im Leerlauf, bis ein Ereignis eintrifft, auf das es reagieren kann, etwa ein Klick auf einen Befehlsknopf. Als Programmierer sind Sie in der Lage, diese Reaktion selbst zu bestimmen. Visual Basic stellt Ihnen hierfür im Codefenster die Liste der möglichen Ereignisse dar.

Doch was ist ein Ereignis in Visual Basic überhaupt?



Bei Microsoft Windows ist so gut wie jede Aktion, auch am Bildschirm, mit einem Ereignis verbunden. Jeder Klick mit der Maus, jeder Tastendruck, die Größenveränderung eines Fensters, alles wird als Ereignis betrachtet. Dabei bedient sich Windows eines Nachrichtensystems, das die Ereignisse weiterreicht.

So wird vom Windows-System die Nachricht *Linke Maustaste gedrückt* an das Fenster oder Steuerelement geschickt, über dem sich die Maus gerade befindet. Beim Erhalt dieser Nachricht löst das Fenster oder das Steuerelement dann bei sich die Ereignisprozedur *Click* aus.

Sie als Visual Basic-Programmierer haben die Möglichkeit, den Inhalt einer Ereignisprozedur zu schreiben, ohne dazu das Handling der Nachrichtenverwaltung kennen zu müssen. Dabei brauchen Sie auch nur die Ereignisse zu behandeln, die in Ihrem Programm eine Bedeutung haben.

Wenn Sie ein Steuerelement in Ihre Form aufnehmen und dieses doppelklicken, sind Ereignisse praktisch das Erste, was Sie an Code von Ihrem Programm sehen. Jedem Ereignis steht eine Ereignisprozedur zur Verfügung, deren Namen mit dem Namen des Objekts, worauf das Ereignis sich bezieht, verknüpft ist. Das kann folgendermaßen aussehen:



```
Private Sub Command1_Click ()  
    'Platz für Ihren Code  
End Sub
```

Hierbei ist *Command1* der Name des Objekts und *Click* der des Ereignisses. In der Klammer stehen bei manchen Ereignissen Argumente, die an die Ereignisprozedur übergeben werden. Diese stehen Ihnen als Variablen im Code der Prozedur zur Verfügung.

Im Code-Fenster (Abbildung 6.16) finden Sie in der linken Listbox *Objekt* die Auswahl der Steuerelemente in Ihrem Formular. Wenn ein Steuerelement in dieser Liste ausgewählt wird, so erscheint in der rechten Listbox *Prozedur* die Übersicht der verfügbaren Ereignisse zu diesem Steuerelement.

Nun wählen Sie mit der Maus für das entsprechende Steuerelement das Ereignis aus, das Sie mit Leben füllen wollen, und schreiben Ihren Code dazu. Beim Auftreten dieses Ereignisses während des Programmlaufs wird Ihr Code dann abgearbeitet.

6.3.1 Ein Steuerelement im Programm ansprechen

Visual Basic vergibt beim Erzeugen eines Objekts automatisch einen Namen, der meist aus einer Abkürzung des Steuerelementtyps und einer laufenden Nummer besteht. Nur über diesen Namen können die Objekte vom Code aus angesprochen werden.

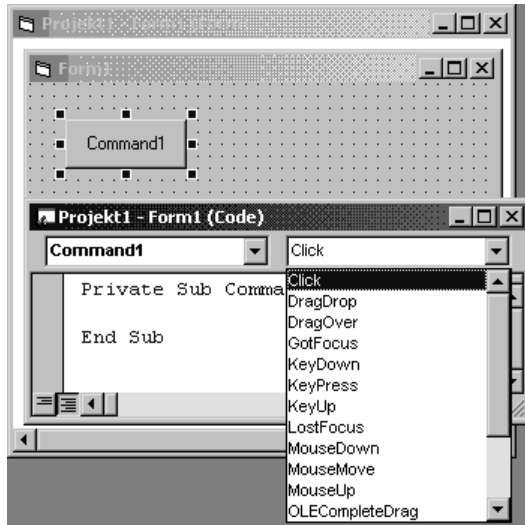


Abbildung 6.16:
Sicht auf das Code-
Fenster mit den
verfügbaren
Ereignissen zum
Steuerelement
»CommandButton
(Befehlsschalt-
fläche)«

Der Dateiname kann sich vom logischen Namen unterscheiden, der im Code verwendet wird. Im Code verwendet man die Bezeichnung, die der Eigenschaft *Name* des Steuerelements gegeben wurde.

Die Methoden und Eigenschaften der Steuerelementobjekte werden nach dem Namen durch einen Punkt getrennt angegeben. Beispielsweise erfolgt die Manipulation der Sichtbarkeit eines Steuerelements über die Eigenschaft *Visible* folgendermaßen:

```
Steuerelement.Visible = True ' Steuerelement ist sichtbar
Command1.Visible = False ' Befehlsschaltfläche ist nicht sichtbar
```

Wenn Sie ein Steuerelement in einem anderen Formular ansprechen wollen, so müssen Sie die Steuerelementangabe um den Formularnamen wie folgt erweitern:

```
Formularnamen.Steuerelement.Visible = False ' Steuerelement ist nicht
sichtbar
Form2.Command1.Visible = False ' Befehlsschaltfläche im Formular 2 ist
nicht sichtbar
```

Auf diese Weise ist ein Steuerelement eindeutig ansprechbar. Eine noch tiefere Strukturierung gibt es nicht. Wenn Sie ein Steuerelement im Code des Formulars ansprechen, so benötigen Sie keinen Formularnamen vor dem Objektnamen.

Ereignisse auswerten

Auch Ereignisse erhalten ihren Namen über den Namen des zugehörigen Steuerelements. Das Ereignis *Click* der Schaltfläche *Command1* heißt beispielsweise *Command1_Click*.



Aber beachten Sie: Manche Steuerelemente können *Click*- und *DbClick* (Doppelklick)-Ereignisse auswerten. Wenn beide Ereignisse verwendet werden, wird beim Ereignis Doppelklick auch immer das Ereignis *Click* miterzeugt und ausgeführt.

In den folgenden Kapiteln werden Ihnen alle wichtigen und häufig benötigten Steuerelemente vorgestellt. In Verbindung mit praxisorientierten, unterschiedlich schweren Projektübungen (Projektaufgaben) erfahren Sie, für welche Aufgaben welche Steuerelemente vorgesehen sind und wie mit ihnen umgegangen bzw. programmiert wird.



Zusätzliche Steuerelemente stehen Ihnen in Abhängigkeit der von Ihnen benutzten Edition von Visual Basic zur Verfügung. Es sind ActiveX-Steuerelemente, die Sie nach Belieben über den Komponentendialog aus dem Menü *Projekt* Ihren Projekten hinzufügen und benutzen können. Welche Steuerelemente in Ihrer Edition verfügbar sind, entnehmen Sie der jeweiligen Dokumentation.

6.4 Das Steuerelement »Form (Formular)«

Die große Frage lautet nun: Welche Applikationen kann ich wie mit Visual Basic realisieren?

In der Regel bestehen Visual Basic-Programme aus mindestens einem Formular (Form). Hier platzieren Sie Ihre Steuerelemente und schreiben Ihren Code dazu. Wie Sie Steuerelemente einfügen, haben Sie bereits kennen gelernt. Ein Formular kann ebenfalls als Steuerelement betrachtet werden.

Das Formular verfügt auch über *Methoden* und *Eigenschaften* und kann *Ereignisse* erzeugen. Es handelt sich hierbei jedoch um eine besondere Art von Steuerelement, das speziell dafür ausgelegt wurde, andere Steuerelemente aufzunehmen und ihnen eine Umgebung zur Existenz zu geben.

Ein Steuerelement allein hat keine Möglichkeit zu existieren. Daher werden Formulare als Behälter für Steuerelemente verwendet, die diesen die Einbindung in die Windows-Umgebung erlauben.

Eigenschaften Eigenschaften sind in Visual Basic für jedes Steuerelement notwendig, somit auch für Formulare. Das Eigenschaftfenster ist ein wichtiger Bestandteil, wenn es um die Manipulation von Steuerelementen geht.

Im Eigenschaftfenster bestimmen Sie das Aussehen des Steuerelements, schalten grundlegende Funktionen frei oder sperren diese und versehen es mit einem Namen, mit dem Sie es später im Code ansprechen.

Die Einstellungen im Eigenschaftfenster bestimmen, wie sich das Steuerelement beim Aufruf im Programm verhält.

Wenn Sie ein Steuerelement in Ihr Formular neu einfügen, so zeigt das Eigenschaftfenster die Grundeinstellung für dieses Element. Dies gilt auch für Formulare.

Ein Formular hat eine ganze Reihe von Grundfunktionen, die ihm das Aussehen und Verhalten eines Windows-Fensters geben.

6.4.1 Übung: Formulare laden und anzeigen

Eine wichtige Eigenschaft von Visual Basic ist die, dass, wenn Sie per Code auf Objekte zugreifen, die noch nicht geladen sind, diese automatisch geladen werden. Sie haben beispielsweise ein Projekt mit zwei Formularen.

Das eine Formular ist sichtbar und geladen, das andere wurde noch nicht angesprochen und demnach auch noch nicht geladen. Das kann nun mit der Anweisung *Load Formname* explizit geschehen.

Beachten Sie aber, dass das Formular noch nicht angezeigt wird. Erst die Methode *Formname.Show* macht das Formular sichtbar. Anders sieht es jedoch beim impliziten Laden des Formulars aus.



Dies geschieht, wenn über den Code des einen Formulars auf eine Eigenschaft oder Methode eines anderen Formulars zugegriffen wird. Es wird dann automatisch das angesprochene Formular nachgeladen.

Angenommen, das Formular *ZweitesFormular* wäre noch nicht geladen und es würde folgende Ereignisprozedur aufgerufen:

```
Private Sub Command1_Click()
    ...'Ihr Code
    ZweitesFormular.Caption = "Test" 'Lädt zweites Formular automatisch
    ...'Ihr Code
End Sub
```

Das implizite Laden eines Formulars entspricht der Anweisung *Load*, d.h., das Formular wird in den Speicher geladen, aber nicht angezeigt.

Das Aussehen eines Formulars lässt sich mit verschiedenen Eigenschaften einstellen.

Sie wollen beispielsweise aus Ihrem Formular ein weiteres laden, das einen festen Rahmen hat:

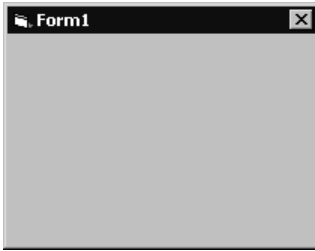
Weitere wichtige Eigenschaften und Methoden von Formularen



Lösung

Die Eigenschaft *BorderStyle* (Abbildung 6.17) bestimmt Aussehen und Funktion des Rahmens (größenveränderbar, fest, kein Rahmen etc.).

Abbildung 6.17:
Eigenschaft *BorderStyle*
setzen



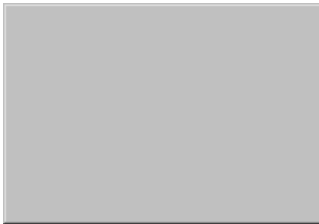
```
Private Sub Command1_Click()  
    ZweitesFormular.BorderStyle=3'Fester Rahmen  
    ZweitesFormular.Show'Neues Fenster anzeigen  
End Sub
```



Oder Sie wollen aus Ihrem Formular ein weiteres laden, das jedoch keinen Titelbalken hat:

Lösung

Abbildung 6.18:
Eigenschaften
ControlBox und
Caption setzen



Die Eigenschaften *ControlBox* und *Caption* beeinflussen die Sichtbarkeit der Titelleiste (Abbildung 6.18).

```
Private Sub Command1_Click()  
    ZweitesFormular.ControlBox=False'Kein Titelbalken  
    ZweitesFormular.Caption=""'Kein Titel  
    ZweitesFormular.Show'Neues Fenster anzeigen  
End Sub
```

Damit die Titelleiste verschwindet, muss die Eigenschaft *Caption* des Formulars auf *Leer* (»«) gesetzt werden. Sie können diese Einstellungen natürlich auch im Eigenschaftenfenster zur Entwicklungszeit vornehmen, wobei Sie das Ergebnis bereits im Formularfenster sehen können.



Sie wollen beispielsweise aus Ihrem Formular ein weiteres laden, das jedoch keine Min/Max-Knöpfe hat:

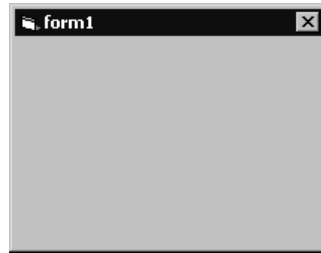
Lösung

Abbildung 6.19:
Eigenschaften *Min-*
Button und *Max-*
Button setzen

Die Eigenschaften *MinButton* und *MaxButton* beeinflussen die Sichtbarkeit der Steuerknöpfe in der Titelleiste eines Fensters und müssen in diesem Beispiel ausgeschaltet (False) sein (Abbildung 6.19).

```
Private Sub Command1_Click()
    ZweitesFormular.MinButton=False'Kein Symbol-Knopf
    ZweitesFormular.MaxButton=False'Kein Vollbild-Knopf
    ZweitesFormular.Show'NeuesFenster anzeigen
End Sub
```

Sie wollen als Nächstes verhindern, dass der Benutzer das Fenster verschieben kann:

**Lösung**

Wenn Sie verhindern wollen, dass der Benutzer das Fenster verschieben kann, so verwenden Sie die Eigenschaft *.Movable* und setzen diese auf *False*. Das Fenster ist somit fest auf dem Bildschirm verankert und lässt sich nicht mehr mit der Maus ziehen. Sie können allerdings noch per Code die Position des Fensters verändern.

```
Private Sub Command1_Click()
    ZweitesFormular.Movable=False'Fenster darf nicht verschoben werden
    ZweitesFormular.Show'NeuesFenster anzeigen
End Sub
```

Eigenschaft
Movable setzen

Wie lässt sich während des Programmlaufs der Inhalt eines Formulars drucken?

**Lösung**

Sie können Ihr Formular auch während des Programmlaufs drucken. Hierfür wird die Methode *PrintForm* bereitgestellt. Diese schickt ein Abbild des Formulars an den Standarddrucker, wobei man keinen Einfluss auf die Druckereinstellung hat. Es werden automatisch die aktuellen Einstellungen des Standarddruckers von Windows verwendet.

Methode **Print-**
Form verwenden

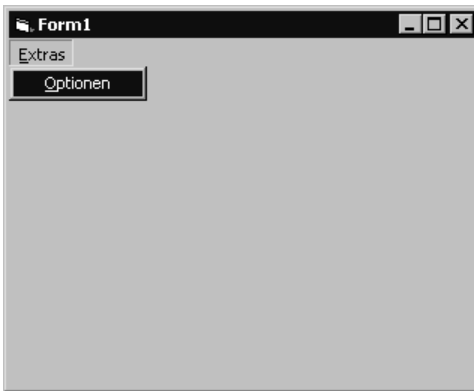
6.4.2 Übung: Unterschiedliche Menütypen im Formular integrieren

Jedes Formular kann ein Menü besitzen. Der Einsatz von Menüs ist aus Anwendungen, die dem Benutzer eine Reihe von Befehlen zur Verfügung stellen, nicht mehr wegzudenken. Mit ihrer Hilfe können Sie Befehle auf übersichtliche Art und Weise thematisch in Gruppen organisieren, so dass der Anwender sie leicht finden und auswählen kann.

Zur Erstellung von Menüs zur Entwurfszeit stellt Visual Basic ein hervorragendes Instrument zur Verfügung, welches in die Entwicklungsumgebung integriert ist und keine Wünsche offen lässt: Menüs werden im *Menü-Editor* aus dem Menü `EXTRAS->MENÜ-EDITOR` erstellt.

Hier geben Sie Ihrem Formular bereits das erste Steuerelement mit. Ein Menü ist ebenfalls ein Steuerelement, das sich allerdings nur in ein Formular einfügen lässt.

Abbildung 6.20:
Menü eingefügt



Erstellen Sie das in Abbildung 6.20 zu sehende Menü *EXTRAS* mit dem Menü-Befehl *OPTIONEN*.

Lösung

Ein Menü hat nur wenige Eigenschaften. Die wichtigsten sind die *Caption* (Überschrift, Inschrift), welche später in der Menüleiste angezeigt wird. Des Weiteren hat das Menü einen Namen *Name*, mit dem das Menü programmintern angesprochen werden soll, in unserer Übung »mnuExtras« (Abbildung 6.21).

Hierbei steht das *mnu* für Menü, also für eine Klassifikation zur besseren Unterscheidung der einzelnen Elemente Ihrer Form und *Extras* ist der optimale mnemonische Begriff für dieses Element.

Weitere Eigenschaften sind: Ein mögliches Tastaturkürzel, die Sichtbarkeit und die Position im Menü.



Abbildung 6.21:
Definition im
Menü-Editor

Einen Menü-Befehl unter das Menü EXTRAS fügen Sie hinzu, indem Sie die Schaltfläche NÄCHSTER betätigen. Da dieser Befehl in der Menügruppe EXTRAS enthalten sein soll, muss er diesem Menü hierarchisch untergeordnet werden. Dies erreichen Sie durch das Aktivieren der Schaltfläche *Pfeil nach rechts* (Abbildung 6.21).

Durch Beenden des Menü-Editors mit der Schaltfläche OK wird die eingegebene Menüstruktur in das Formular übernommen. Da die Menüstruktur im Entwicklungsmodus bereits sichtbar ist, können Sie Ihre Menüangaben sofort, ohne Kompilierung des Programms, überprüfen.

Unsere Applikation soll ein zweites Formular aufrufen, wenn der Befehl *Optionen* aus dem Menü EXTRAS ausgelöst wird.



Lösung

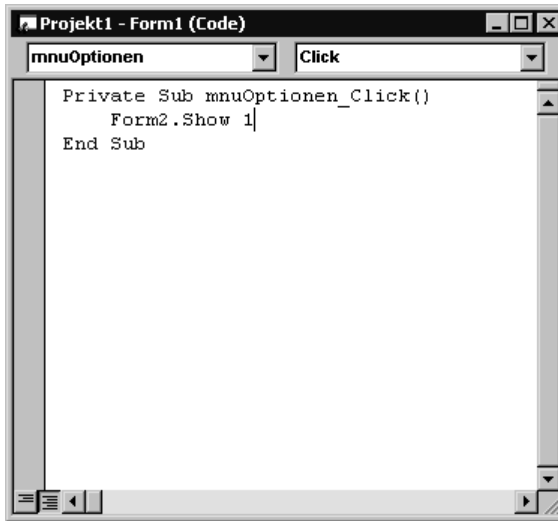
Jeder Menü-Befehl kann das Ereignis *Click* auslösen. Wenn Sie mit der Maus den Menü-Befehl im Formularfenster anklicken, wird das Code-Fenster geöffnet.

Hier hinterlegen Sie Ihre anwendungsspezifischen Anweisungen. In unserem Falle (Abbildung 6.22) ausschließlich die Anweisung `Form2.Show 1` zum Laden und Anzeigen eines weiteren Formulars.

Der Menü-Editor lässt sich auch auf dem Formular mit der rechten Maustaste über das Kontextmenü aufrufen. Dieses Menü gewährt auch Zugriff auf das Fenster für die Darstellung der Eigenschaften von Steuerelementen.



Abbildung 6.22:
Ereignisprozedur
des Menü-Befehls
Optionen

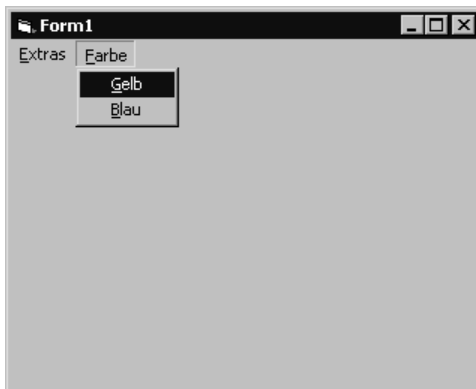


```
Private Sub mnuOptionen_Click()  
    Form2.Show 1  
End Sub
```



Um einen weiteren Menütitel in der Menüleiste hinzuzufügen, rufen Sie wieder den Menü-Editor auf. Es soll das Menü FARBE mit den Menü-Befehlen Gelb und Blau erstellt werden (Abbildung 6.23).

Abbildung 6.23:
Zusätzlicher Menü-
titel mit Menü-
Befehlen



Lösung

Öffnen Sie erneut den Menü-Editor und definieren Sie das zusätzlich gewünschte Menü, wie in Abbildung 6.24 zu sehen. Aus dieser Vorgehensweise folgt:

- ▶ Menü-Steuerelemente, die im Listefeld des Menü-Editors linksbündig ausgerichtet sind, werden in der Menüleiste als Menütitel angezeigt (wie die Menüs EXTRAS und FARBE).

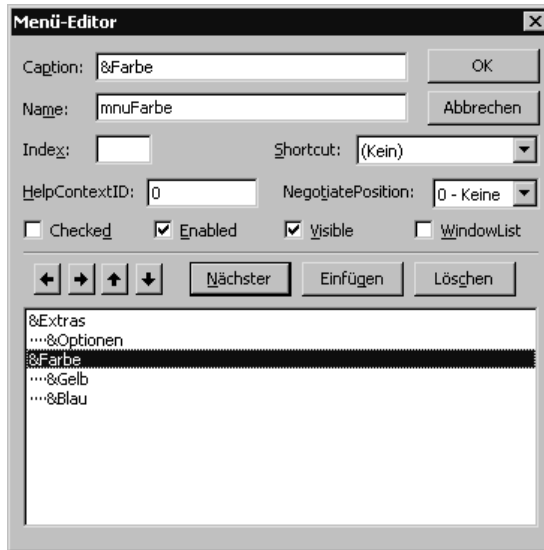


Abbildung 6.24:
Im Menü-Editor
definieren

- ▶ Menü-Steuerelemente, die um eine Position eingerückt sind, werden als Menü-Befehle des linksbündig ausgerichteten Menütitels angezeigt (wie die Menü-Befehle *Optionen*, *Gelb* und *Blau*).
- ▶ Menü-Steuerelemente, die um zwei Positionen eingerückt sind, machen eine einfach eingerückte Position zu einem Untermenütitel, die um zwei Positionen eingerückten Menü-Steuerelemente zu Elementen dieses Untermenütitels. In einem solchen Fall spricht man auch von so genannten »Kaskaden-Menüs«.
- ▶ Menü-Steuerelemente, die nur mit einem Bindestrich '-' in der Caption-Eigenschaft versehen sind, werden als Trennlinien zur Unterteilung eines Menüs in logische Gruppen interpretiert.

Eine Trennlinie in ein Menü einfügen

Erzeugen Sie im Menü EXTRAS zwei weitere Menü-Befehle (*Schrift* und *Korrektur*) und heben Sie diese vom Befehl *Optionen* durch eine Trennlinie ab (Abbildung 6.25).



Lösung

Markieren Sie im Menü-Editor die Zeile mit dem Eintrag *Farbe* und klicken Sie anschließend auf die Schaltfläche EINFÜGEN.

Positionieren Sie die Einfügemarke auf dem Feld *Caption* und geben Sie einen Bindestrich '-' ein. Unter *Name* hinterlegen Sie z.B. "mnuTrenn« (Abbildung 6.26). Analog dieser Vorgehensweise fügen Sie die neuen Menü-Befehle *Schrift* und *Korrektur* ein.

Abbildung 6.25:
Trennlinie im Menü

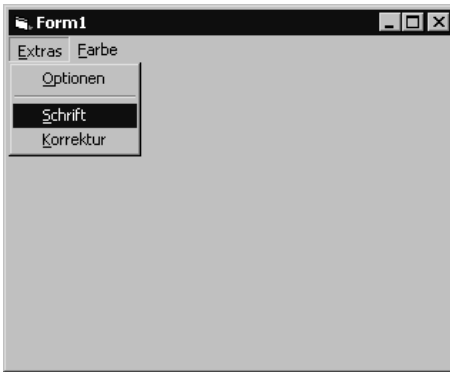


Abbildung 6.26:
Trennlinie einfügen



Schließen Sie zur Überprüfung des Ergebnisses den Menü-Editor durch einen Mausklick auf die Schaltfläche OK, worauf Visual Basic wieder in die Form Ihres Beispielprogrammes zurückkehrt.

Kaskaden-Menü erstellen

Nachdem Sie sich von dem hoffentlich korrekten Outfit Ihrer Menüs überzeugt haben, rufen Sie den Menü-Editor erneut auf. Wir wollen nun noch eine weitere Variante integrieren, die das Menüentwurf fenster zu bieten hat, nämlich die Entwicklung eines Kaskadenmenüs.



In unserem Beispiel bietet es sich an, aus den Menü-Befehlen *Gelb* und *Blau* des Menüs *FARBE* einen Untermenütitel *Farbe* im Menü *EXTRAS* zu machen, dem dann die einzelnen auswählbaren Farben als Untermenü-Befehle hinzugefügt werden (Abbildung 6.27).

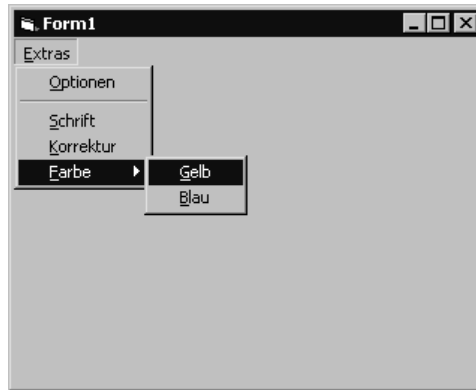


Abbildung 6.27:
Kaskaden-Menü erzeugt

Lösung

Um dies zu erreichen, müssen Sie nur die Menü-Elemente *Farbe*, *Gelb* und *Blau* im Menü-Editor um jeweils eine Position einrücken. Positionieren Sie den Cursor hierzu auf dem jeweiligen Menüeintrag und klicken Sie dann auf die Schaltfläche mit dem nach rechts zeigenden Pfeil (Abbildung 6.28).



Abbildung 6.28:
Kaskaden-Menü erstellen

Schließen Sie nun den Menü-Editor durch Anklicken der Schaltfläche OK und vergleichen Sie das Ergebnis mit der Abbildung 6.27.

Wie Sie sehen können (Abbildung 6.27), hat Visual Basic hinter den Befehl *Farbe* einen kleinen Pfeil platziert, der Sie darauf hinweisen soll, dass es sich bei diesem Menü-Befehl um einen Untermenütitel handelt, der noch weitere Unterelemente (Befehle) enthält.

Menü-Code eingeben

Im Folgenden wollen wir an einer Übung die Vorgehensweise demonstrieren, wie noch funktionslose Menüs oder deren Befehle mit Leben gefüllt werden können. Das Programm, das übrigens schon zu diesem Zeitpunkt lauffähig ist, soll den Formularhintergrund über die entsprechenden Menü-Befehle entweder gelb oder blau darstellen.

Lösung

Öffnen Sie hierzu im Formular das Menü EXTRAS durch Mausklick und klicken Sie anschließend auf den Unterbefehl *Gelb* des Befehls *Farbe*, woraufhin sich die Ereignis-Prozedur *mnuGelb_Click()* öffnet.

```
Private Sub mnuGelb_Click()  
    Form1.BackColor = &HFFFF&  
End Sub
```

In diese Prozedur setzen Sie nun die Eigenschaft *BackColor* des aktiven Formulars *Form1* auf den Farbwert Gelb.



Den zu einer Farbe zugehörigen Farbwert erhalten Sie u. a. durch das Auswählen einer Farbe im Eigenschaftenfenster in der Eigenschaft *BackColor*.

Schließen Sie jetzt das Fenster durch Doppelklick auf das Systemmenüfeld.

Um zu testen, ob Visual Basic korrekt auf das Click-Ereignis reagiert, starten Sie nun das Programm durch Drücken der Taste **F5** bzw. durch Anklicken des entsprechenden Symbols und wählen dann über den Befehl *Farbe* des Menüs EXTRAS den Unterbefehl *Gelb* aus.

Analog dieser Vorgehensweise füllen Sie den Befehl *Blau* mit Leben.

```
Private Sub mnuBlau_Click()  
    Form1.BackColor = &HFF0000  
End Sub
```



Daraus folgt, dass jeder Menü-Befehl im Menü mit einer eigenen Click-Ereignisprozedur ausgestattet ist.

Menüs deaktivieren und ausblenden

Da die Menü-Befehle *Schrift* und *Korrektur* noch keine Funktionalität aufweisen, sollen diese auch optisch erkennbar sein, indem sie als inaktiv gekennzeichnet angezeigt werden (Abbildung 6.29).

Lösung

Öffnen Sie hierzu wieder den Menü-Editor und markieren Sie den Menü-Befehl *Schrift*. Klicken Sie nun auf das Kontrollkästchen *Enabled*. Wiederholen Sie diese Schritte ebenfalls für den Menü-Befehl *Korrektur*.

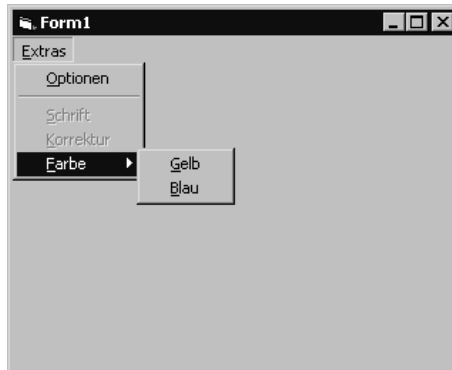


Abbildung 6.29:
Menü-Befehle
deaktivieren

Nach Verlassen des Menü-Editors sollte Ihr Formular nach dem Öffnen des Menüs EXTRAS wie in Abbildung 6.29 aussehen. Die Menü-Befehle *Schrift* und *Korrektur* des Menüs EXTRAS müssen in Hellgrau dargestellt sein und dürfen auf das Anklicken mit der Maus nicht reagieren.

6.4.3 Übung: Im Formular Freihand zeichnen



Es soll die Möglichkeit geschaffen werden, in einem Formular Freihand zeichnen zu können. Dabei ist der Mauszeiger wie ein Malstift zu verwenden und somit jede Mausbewegung in ein »Kritzeln«, wie in Abbildung 6.30 zu sehen, zu übernehmen.

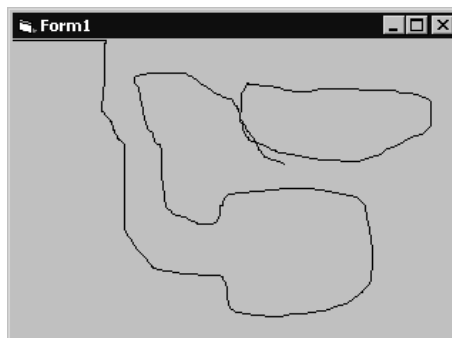


Abbildung 6.30:
Freihand zeichnen
im Formular

Lösung

Wird der Mauszeiger über das Formular bewegt, so hat dies das Auslösen des Ereignisses *MouseMove* zur Folge. Somit sind sämtliche Aktionen, die das Zeichnen betreffen, in diese Prozedur zu legen.

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Line (Form1.CurrentX, Form1.CurrentY)-(X, Y), QBColor(0)
End Sub
```

Methode *Line* Mit Hilfe der Methode *Line* lassen sich sowohl Linien als auch Rechtecke zeichnen.

Syntax `object.Line Step (x1,y1) - Step (x2,y2), color, BF`

Erläuterung der Line-Parameter:

Object Optionaler Parameter, der angibt, auf welches Objekt sich die Methode bezieht. Wird er weggelassen, so wird angenommen, dass das Objekt, welches den Fokus besitzt, das Zielobjekt ist.

Step Ein optionales Schlüsselwort, welches angibt, dass die angegebenen Startkoordinaten (x1, y1) relativ zur momentanen Grafikposition sind. Diese kann über die Eigenschaften *CurrentX* und *CurrentY* ermittelt werden.

(X1, Y1) Optionaler Parameter, der die Startposition der Linie angibt. Wird diese Angabe weggelassen, startet die Linie an der momentanen Grafikposition.

Step Ein optionales Schlüsselwort, welches angibt, dass die angegebenen Endkoordinaten (x2, y2) relativ zur momentanen Grafikposition sind.

(X2, Y2) Erforderlicher Parameter, der die Endposition der Linie angibt.

Color Optionaler Parameter, der die RGB-Farbe der Linie angibt. Wird er weggelassen, bestimmt die *ForeColor*-Eigenschaft die Farbe.

B Optionaler Parameter, der bewirkt, dass anstelle einer Linie ein Rechteck gezeichnet wird. Die Parameter (x1, y1) und (x2, y2) bestimmen in diesem Fall die gegenüberliegenden Ecken des Rechtecks. Wird der Parameter *F* nicht angegeben, so wird das Rechteck abhängig vom Wert der Eigenschaften *FillColor* und *FillStyle* gefüllt.

F Optionaler Parameter, der nur zusammen mit dem Parameter *B* benutzt werden kann. Er bewirkt, dass das Rechteck mit der gleichen Farbe gefüllt wird, mit der auch dessen Umrisse gezeichnet wurden.



Nach Ausführung der Methode werden die Eigenschaften *CurrentX* und *CurrentY* auf die im Parameter (x2, y2) angegebenen Werte gesetzt.

Die Koordinaten, wo gezeichnet werden soll, werden automatisch beim Bewegen der Maus an das Ereignis *MouseMove* als Parameter übergeben (X As Single, Y As Single).

Mit der Funktion *QBColor* arbeiten Die Funktion *QBColor* gibt den RGB-Wert einer Farbe, die über eine Nummer spezifiziert wird, als Long-Wert zurück.

Syntax `QBColor (color)`

Erläuterung der Parameter:

Color Nummer der Farbe, deren RGB-Wert ermittelt werden soll, dabei entspricht

0 = Schwarz 8 = Grau

1 = Blau 9 = Hellblau

2 = Grün	10 = Hellgrün
3 = Cyan	11 = Hellcyan
4 = Rot	12 = Hellrot
5 = Magenta	13 = Hellmagenta
6 = Gelb	14 = Hellgelb
7 = Weiß	15 = Hellweiß

```
Form1.BackColor = QBColor(4)
```

Diese Anweisung setzt die Hintergrundfarbe des Formulars *Form1* auf die Farbe Rot.



6.4.4 Übung: Linien gezielt im Formular zeichnen

Damit nicht grundsätzlich bei jeder Mausbewegung im Formular »gekritzelt« wird, soll das Freihandzeichnen erst in Verbindung mit dem Drücken und Festhalten der linken Maustaste aktiviert werden. Dabei ist dann wieder der Mauszeiger wie ein Malstift zu verwenden und somit jede Mausbewegung in ein »Kritzeln«, wie in Abbildung 6.31 zu sehen, zu übernehmen.

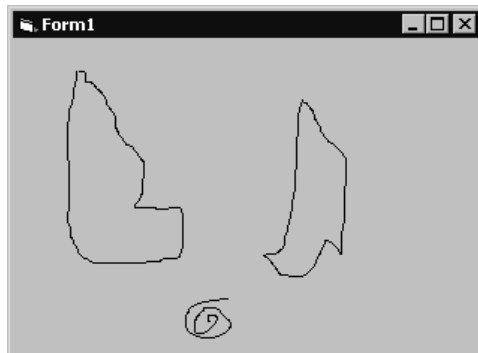


Abbildung 6.31:
Gezielt zeichnen

Lösung

Wird der Mauszeiger über das Formular bewegt, so hat dies das Auslösen des Ereignisses *MouseMove* zur Folge. Somit sind sämtliche Aktionen, die das Zeichnen betreffen, in diese Prozedur zu legen.

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If Button = 1 Then
        Line (Form1.CurrentX, Form1.CurrentY)-(X, Y), QBColor(0)
    End If
End Sub
```

Da ein Zeichnen nur dann erfolgen soll, wenn die linke Maustaste gedrückt und festgehalten wird, muss diese irgendwie abgefragt werden können. Aus der Variablen *Button*, die der Ereignisprozedur *MouseMove* übergeben wird, lässt sich auslesen, welche Maustaste betätigt wurde. Hat die Integervariable *Button* z.B. den Wert *1*, so ist die linke Maustaste gedrückt worden.

Das Zeichnen über die Methode *Line* ist somit über die Variable *Button* zu bedingen, also erst dann wie gewünscht auszuführen.

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    Form1.CurrentX = X
    Form1.CurrentY = Y
End Sub
```

Damit auch genau an der Stelle im Formular, an der die Maustaste gedrückt und festgehalten wird, das »Kritzeln« erfolgt, muss die Mausposition gemerkt werden. Dazu bemühen wir die Ereignisprozedur *MouseDown*, in der die Mauskoordinaten den Eigenschaften *CurrentX* und *CurrentY* der Form zu übergeben sind.

CurrentX und CurrentY

Mit *CurrentX* legen Sie die horizontalen und mit *CurrentY* die vertikalen Koordinaten für die nächste Grafikmethode im Formular fest.



6.4.5 Übung: Formular bildschirmmittigg positionieren

Es soll eine globale Prozedur geschaffen werden, die es ermöglicht, unabhängig von der Bildschirmauflösung beliebige Formulare bei Start (Abbildung 6.32) in der Bildschirmmitte zu positionieren.

Lösung

Wird eine Applikation mit nur einem Formular gestartet, so erscheint das Formular immer und grundsätzlich an der Position auf dem Bildschirm, an der es innerhalb der Visual Basic-Entwicklungsumgebung erstellt wurde. Auch ein Verschieben des Fensters mit nachfolgendem Neustarten der Applikation ändert nichts an dieser Tatsache.

Da eine globale Prozedur zu schaffen ist, die für eine unbekannte Anzahl Formulare und nicht statisch für ein Formular, z.B. *Form1*, gelten soll, ist über das Menü PROJEKT ein Modul, *BAS*-Modul, dem aktuellen Visual Basic-Projekt hinzuzufügen (Abbildung 6.33).

In diesem Modul erstellen wir eine eigene Prozedur mit Namen *Form_Zentrieren*, die für uns das bildschirmmittigg Positionieren eines Formulars erledigt.

Parameter

Da die Prozedur nicht nur ein bestimmtes Formular, sondern global, also vollkommen variabel Formulare positionieren soll, muss ein Parameter eingeführt werden. In der Formvariablen *F* ist der Name der Form zu übergeben, die in der Mitte des Bildschirms zu zentrieren ist.

Das Steuerelement »Form (Formular)«

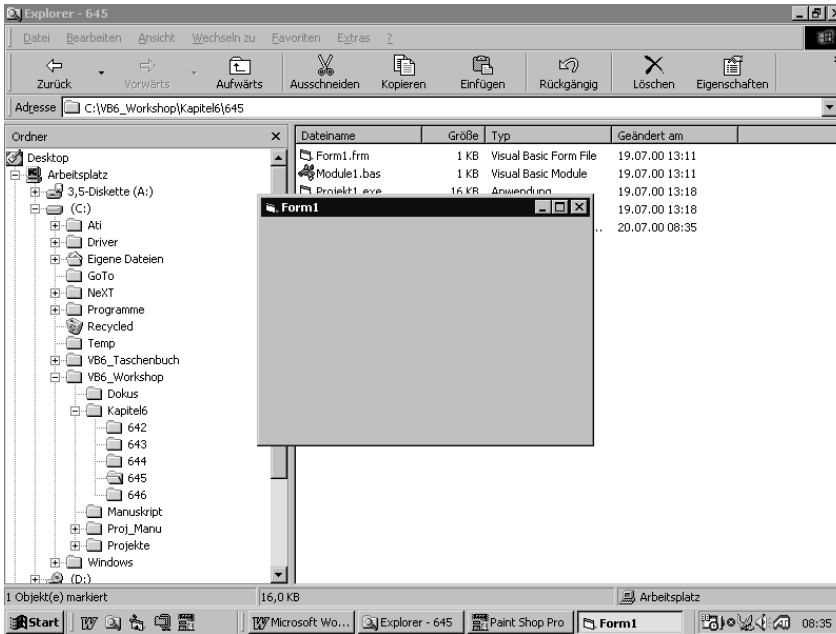


Abbildung 6.32:
Formular zentrieren

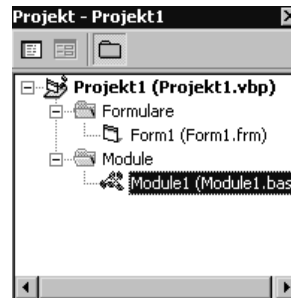


Abbildung 6.33:
BAS-Modul
einfügen

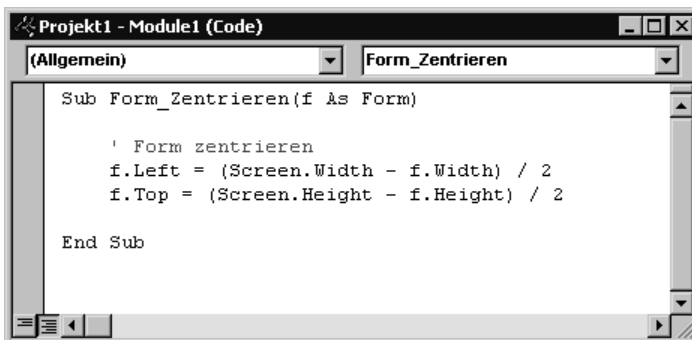


Abbildung 6.34:
Prozedur Zentrieren

Anschließend werden die Formkoordinaten (Position vom linken Rand und Position vom oberen Rand), abhängig von der Bildschirm- (Screen-Breite und -Höhe) und der Formulargröße (Form-Breite und -Höhe) neu berechnet.

```
Private Sub Form_Load()  
    Call Form_Zentrieren(Form1)  
End Sub
```

Damit bei Applikationsstart oder beim Laden eines beliebigen Formulars auch auf das Zentrieren verwiesen wird, ist die Ereignisprozedur *Load* des Formulars zu erweitern.

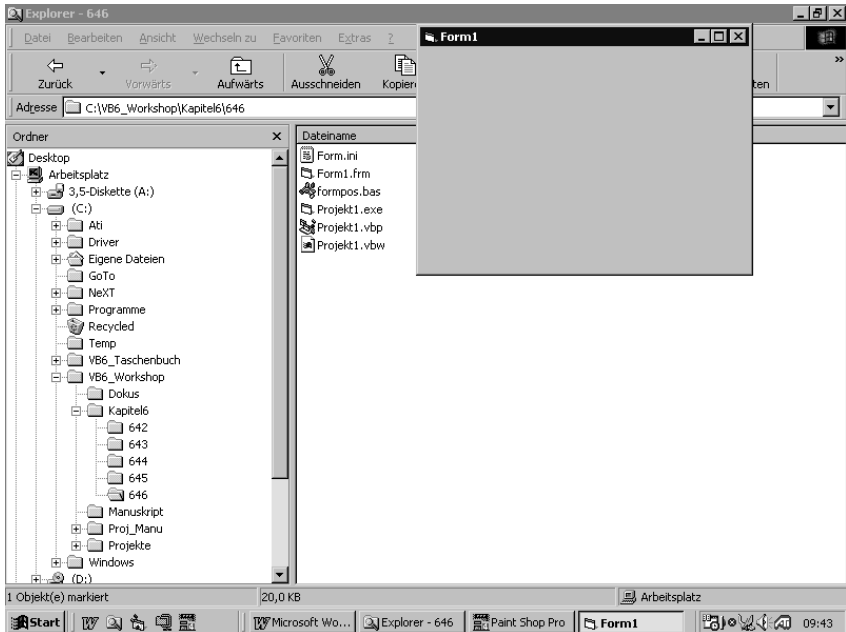
Form_Load ist die Prozedur, die noch vor dem eigentlichen Anzeigen des Formulars auf dem Bildschirm aufgerufen wird. In dieser ist der Aufruf unserer Zentrier-Prozedur *Form_Zentrieren(Form1)* mit Übergabe des Formularnamens zu integrieren.



6.4.6 Übung: Formular wie bei letzter Benutzung positionieren

Es soll eine globale Prozedur geschaffen werden, die es ermöglicht, unabhängig von der Bildschirmauflösung beliebige Formulare bei Start (Abbildung 6.35) an der Position der letzten Benutzung auf dem Bildschirm zu positionieren.

Abbildung 6.35:
Formular wie bei
letzter Benutzung
positionieren



Dadurch wird die Formpositionierung anwenderfreundlicher und eine Form nicht wie meist üblich mittig am Bildschirm ausgerichtet, sondern so, wie der Anwender sein Formular zuletzt verlassen hat. Der Anwender kann somit die Formularposition applikationsabhängig einstellen.

Lösung

Wird eine Applikation mit einem Formular gestartet, so erscheint das Formular immer und grundsätzlich an der Position auf dem Bildschirm, an der es innerhalb der Visual Basic-Entwicklungsumgebung erstellt wurde, und nicht an der Position der letzten Benutzung.

Da wieder eine globale Prozedur zu schaffen ist, ist über das Menü PROJEKT ein Modul, *BAS-Modul*, dem aktuellen Visual Basic-Projekt hinzuzufügen. In diesem Modul erstellen wir zwei eigene Prozeduren, die für uns die definierten Anforderungen erledigen.

Die Prozedur *Pos_Write* schreibt die Koordinaten, an der eine Form zuletzt positioniert war, in eine sequenzielle Datei (»normale« ASCII/ANSI-Textdatei, die mit jedem beliebigen Editor geöffnet und gelesen werden kann (Abbildung 6.36)).

Prozedur
Pos_Write

```
Sub Pos_Write (DN$, FoNa As Form)
    ' Freie Dateinummer holen
    Dim dateinr As Integer
    dateinr = FreeFile
    ' Formposition in Positionsstring stellen
    POS$ = Space$(15)
    Mid$(POS$, 1, 6) = FoNa.Left
    Mid$(POS$, 8, 6) = FoNa.Top
    ' Formposition speichern
    Open DN$ For Output As dateinr
        Print #dateinr, POS$
    Close dateinr
End Sub
```

```
Call Pos_Write (Dateiname$, Formname)
```

Syntax

Dateiname, Typ String: Name der Datei, in der die Formkoordinaten gespeichert werden sollen. Die Angabe von Laufwerk und Pfad ist optional.

Parameter

Formname, Typ Form: Name der Form, deren Bildschirmkoordinaten interpretiert und in eine Datei gesichert werden sollen.

```
Private Sub Form_Unload(Cancel As Integer)
    Call Pos_Write ("C:\test.ini", form1)
End Sub
```



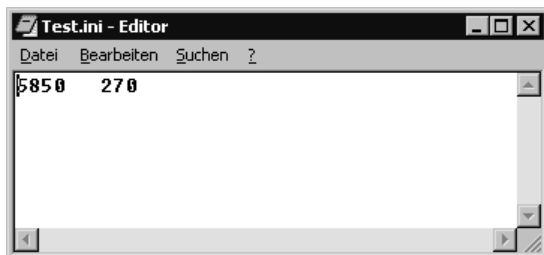
Die Koordinaten der aktiven Startform einer Applikation sollen beim Beenden des Programms gespeichert werden (Abbildung 6.36), um dieses Formular bei erneutem Aufruf der Applikation genau an derselben Stelle wie beim Verlassen des Formulars auf dem Bildschirm positionieren zu können. Dazu bemühen wir die Ereignisprozedur *Form_Unload*.

Die Koordinaten der letzten Bildschirmposition des Formulars *Form1* werden in die Datei *Test.ini* von Laufwerk *C:* geschrieben (Abbildung 6.36). Dadurch wird

Bemerkung

die Formpositionierung anwenderfreundlicher, da mit Hilfe der nachfolgenden Prozedur *Pos_Read* das Formular so am Bildschirm ausgerichtet wird, wie es zuletzt verlassen wurde.

Abbildung 6.36:
Mit dem Windows-
Editor »Notepad«
die gesicherten
Formkoordinaten
ansetzen



Prozedur Pos_Read

Die Prozedur *Pos_Read* liest die Koordinaten, an der eine Form zuletzt positioniert war, aus einer sequenziellen Datei aus.

```
Sub Pos_Read (DN$, FoNa As Form)
    ' Freie Dateinummer ermitteln
    Dim dateinr As Integer
    dateinr = FreeFile
    ' Formposition aus Positionsstring laden
    Open DN$ For Input As dateinr
        Line Input #dateinr, POS$
    Close dateinr
    ' Form wie zuletzt positionieren
    FoNa.Left = Val(Trim$(Mid$(POS$, 1, 6)))
    FoNa.Top = Val(Trim$(Mid$(POS$, 8, 6)))
End Sub
```

Syntax Call Pos_Read (Dateiname\$, Formname)

Parameter *Dateiname*, Typ String: Name der Datei, in der die Formkoordinaten gespeichert sind. Die Angabe von Laufwerk und Pfad ist optional.

Formname, Typ Form: Name der Form, die entsprechend der gelesenen Koordinaten auf dem Bildschirm neu positioniert werden soll.



```
Private Sub Form_Load()
    Call Pos_Read ("C:\test.ini", form1)
End Sub
```

Die Startform einer Applikation soll an der Stelle auf dem Bildschirm positioniert werden, wie Sie beim Beenden der Applikation verlassen wurde. Dazu bemühen wir die Ereignisprozedur *Form_Load*.

Bemerkung Die Koordinaten der letzten Bildschirmposition des Formulars *Form1* werden aus der Datei *Test.ini* von Laufwerk *C:* gelesen und das Formular *Form1* wieder genau an dieser Stelle positioniert.

6.5 Das Steuerelement »CommandButton«

Die Befehlsschaltfläche (CommandButton) stellt eine Standard-Befehlsschaltfläche von Windows dar. Mit ihr können Aktionen ausgelöst werden, beispielsweise der Start oder der Abbruch eines Prozesses. Das Aussehen ist durch Eigenschaften beeinflussbar.

Die Befehlsschaltfläche ist ein wichtiges Steuerelement beim Entwurf von grafischen Oberflächen. Sie dient zum Auslösen einer Aktion oder zum Bestätigen eingegebener Daten, die beispielsweise über eine *TextBox* eingegeben werden können.

6.5.1 Übung: Auf Anforderung einen Button ein- oder ausblenden

Über eine zweite Befehlsschaltfläche (CommandButton) soll es ermöglicht werden, die Befehlsschaltfläche Command1 ein- oder auszublenden (Abbildung 6.37), sprich als sichtbar oder nicht sichtbar darzustellen. Diese Aktion soll über Maus oder Tastaturkürzel angewählt werden können.



Abbildung 6.37:
Command1 wiederum einblenden

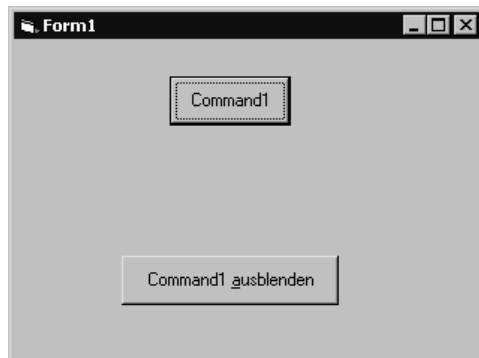


Abbildung 6.38:
Command1 ausblenden

Wie in der Abbildung 6.38 zu sehen ist, müssen je nach Status, ob der erste Button ein- oder ausgeblendet ist, der Text und das Tastaturkürzel im zweiten Button variabel versorgt werden.

Lösung

Damit bei Mausklick auf die zweite Befehlsschaltfläche der erste Button tatsächlich nicht mehr sichtbar wird, ist die Ereignisprozedur *Click* des Command-Buttons *Command2* zu erweitern.

```
Private Sub Command2_Click()  
    Command2.Caption = "Command1 &einblenden"  
    Command1.Visible = False  
End Sub
```

Das wichtigste Ereignis der Befehlsschaltfläche ist wohl das *Click*-Ereignis. Es erfüllt den Hauptzweck einer Befehlsschaltfläche, nämlich dem Benutzer die Möglichkeit zu geben, etwas zu bestätigen, eine Aktion zu starten oder abzubauen.

Mit der Eigenschaft *Visible* eines Steuerelements kann bestimmt werden, ob das angesprochene Steuerelement im Formular als sichtbar oder unsichtbar erscheinen soll. So ist in unserem Beispiel die Befehlsschaltfläche *Command1* in der Eigenschaft *Visible* auf *False* zu setzen.

Darüber hinaus kann die Befehlsschaltfläche auch auf Tastatureingaben reagieren. Es muss ihr nur ein Tastaturkürzel zugeordnet werden.

```
Command2.Caption = "Command1 &einblenden"
```

Hierzu kann man einen Buchstaben der *Caption*-Eigenschaft unterstreichen, wie Sie es sicherlich schon in anderen Anwendungen gesehen haben. Fügen Sie einfach ein *&*-Zeichen vor dem Buchstaben ein, den Sie unterstreichen wollen.

Das *&*-Zeichen wird im Objekt nicht angezeigt, sondern als ein Steuerzeichen zum Unterstreichen des nachfolgenden Buchstabens interpretiert. Wenn Sie während des Programmlaufs nun diesen Buchstaben in Verbindung mit der **ALT**-Taste drücken, so wird dies wie ein *Click*-Ereignis behandelt.

```
Dim X As Integer  
Private Sub Command2_Click()  
    If X = 0 Then  
        X = 1  
        Command2.Caption = "Command1 &einblenden"  
        Command1.Visible = False  
        Exit Sub  
    End If  
    If X = 1 Then  
        X = 0  
        Command2.Caption = "Command1 &ausblenden"
```

```

    Command1.Visible = True
End If
End Sub

```

Damit eingeblendet werden kann, wenn ausgeblendet wurde und umgekehrt, ist es notwendig, eine Variable *Global* zu definieren, über die der jeweilige Sichtbarkeits-Status gemerkt und gesteuert werden kann. Starten Sie nun diese Applikation und aktivieren Sie den zweiten Button, so enthält die Variable *X* je nachdem, ob der erste Button ein- oder ausgeblendet ist, den Wert 1 oder 0.

6.5.2 Übung: Button mit Grafik und QuickInfo

Es soll, wie in Abbildung 6.39 zu sehen, eine Befehlsschaltfläche diesmal mit integrierter Grafik (C:\windows\Kugeln.Bmp) und erklärender QuickInfo (Bildprogramm starten) im Formular visualisiert werden.

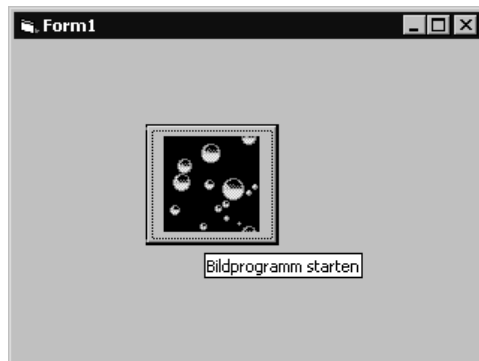


Abbildung 6.39:
Button mit Grafik
und QuickInfo

Lösung

Weitere Eigenschaften der Befehlsschaltfläche betreffen unter anderem ihre optische Gestaltung. So können Sie sich zum Beispiel Bilder in der Schaltfläche anzeigen lassen.

Um das zu erreichen, muss jedoch die Eigenschaft *Style* auf *Grafisch* gesetzt werden (Abbildung 6.40). Über die Eigenschaft *Picture* können dann Bilder vom Typ *.bmp*, *.pcx*, *.wmf*, *.gif* und *.jpg* in die Befehlsschaltfläche geladen und visualisiert werden.

Seit Visual Basic 5.0 können die meisten Steuerelemente mit einem *ToolTipText* versehen werden (Abbildung 6.41). Hierbei handelt es sich um einen erklärenden Text, der angezeigt wird (Abbildung 6.39), wenn Sie mit dem Mauszeiger über einem Steuerelement (hier der Befehlsschaltfläche) eine kurze Zeit verweilen.

Abbildung 6.40:
Eigenschaft Style
verändern



Abbildung 6.41:
Eigenschaft Tool-
TipText setzen



6.5.3 Übung: Der Button hüpft weg!

Diese Übung bietet mal zur Abwechslung eine spielerische Variante. Eine Applikation soll über die Befehlsschaltfläche *Beenden* beendet werden, kann aber nicht, weil die Befehlsschaltfläche immer dann, wenn man mit dem Mauszeiger über ihr ist, weghüpft und somit ein Beenden nicht zulässt (Abbildungen 6.42 und 6.43).

Lösung

Wird der Mauszeiger über den Button bewegt, so hat dies das Auslösen des Ereignisses *MouseMove* zur Folge. Somit sind sämtliche Aktionen, die das »Weghüpfen« betreffen, in diese Prozedur zu legen.

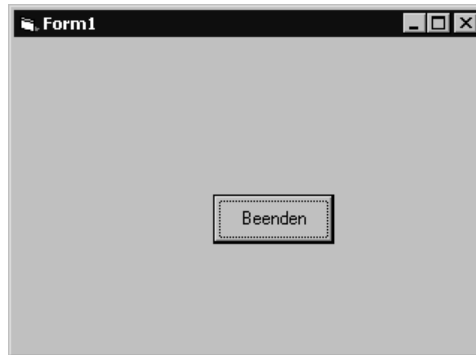


Abbildung 6.42:
Button an Start-
Position



Abbildung 6.43:
Button ist
weggehüpft

```
Private Sub Command1_MouseMove(Button As Integer, Shift As Integer, X
As Single, Y As Single)
    If Command1.Left = 2000 Then
        Command1.Left = 0
    Else
        Command1.Left = 2000
    End If
End Sub
```

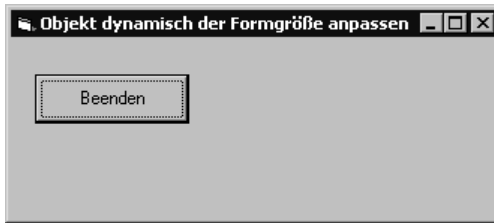
Ist die Befehlsschaltfläche an der Startposition ($Left = 2000$), so wird die Eigenschaft *Left* (Entfernung vom linken Rand) auf 0 gesetzt, andernfalls wieder mit dem Startwert 2000 initialisiert. Somit hüpfert der Button immer von Position 2000 zu Position 0 und wieder zurück.

6.5.4 Übung: Buttongröße dynamisch zur Formgröße anpassen

In sehr vielen Windows-Applikationen ist zu sehen, dass sich die Größe der im Formular eingebetteten Steuerelemente (Objekte) nicht proportional der Formgröße anpasst, wenn diese sich ändert. Und so stehen oft die Objekte quasi »verloren« im Raum, nicht vergrößert, nicht gezoomt, wenn z. B. das Formular als Vollbild dargestellt oder aufgezogen wird.



Abbildung 6.44:
Bei Veränderung
der Formulargröße
ändert sich auch
die Größe der
Befehlsschaltfläche



Am Beispiel einer Befehlsschaltfläche soll nun eine Lösung erarbeitet werden, wie sich die Größe dieses Objekts an der Größe des Formulars dynamisch orientiert, d.h. wie die Befehlsschaltfläche größer dargestellt wird, wenn sich das Formular vergrößert, und kleiner dargestellt wird, wenn sich das Formular verkleinert (Abbildungen 6.44 und 6.45).

Abbildung 6.45:
Bei Veränderung
der Formulargröße
ändert sich auch
die Größe der
Befehlsschaltfläche



Lösung

Damit später überhaupt eine dem Verhältnis entsprechende Größenanpassung bei Veränderung der Formulargröße erfolgen kann, muss zuerst die Proportion Befehlsschaltflächengröße zu Formulargröße im Ur-Originalzustand berechnet werden.

```
Private Sub Form_Load()  
    ' Verhältnis zueinander berechnen  
    höhe = Form1.Height / Command1.Height  
    breite = Form1.Width / Command1.Width  
End Sub
```

Dies geschieht in der Ereignisprozedur *Form_Load*. Das Größenverhältnis wird in den beiden als global zu definierenden Variablen *Höhe* und *Breite* gemerkt.


```
Private Sub Form_Resize()
    Command1.Height = Form1.Height / höhe
    Command1.Width = Form1.Width / breite
End Sub
```

Damit sich die Größe der Befehlsschaltfläche proportional zur Fenstergröße anpasst, wenn sich diese ändert, sind immer dann die Eigenschaften *Height* und *Width* des Buttons neu zu setzen. Das Einzige, was uns jetzt noch fehlt, ist der Auslöser, das Ereignis, mit dem wir in Visual Basic merken, dass sich die Formulargröße verändert hat.

Diese Veränderung bekommen wir über das Ereignis *Form_Resize* mitgeteilt, das auftritt, wenn das Formular zum ersten Mal angezeigt wird oder wenn sich der Fensterstatus (z.B. beim Maximieren, Minimieren, Wiederherstellen) oder ganz allgemein die Größe des Formulars ändert. So ist genau in diese Prozedur das Setzen der neuen Befehlsschaltflächengröße zu implementieren.

Form_Resize

6.5.5 Übung: Individuelle Programmschnellstartleiste (Application Caller)



Da wir auf unserer Festplatte meist sehr viele Programmpakete (z.B. Datenbank, Textverarbeitung, Spiele, Compiler etc.) in unterschiedlichen Verzeichnissen abgelegt haben, artet z.B. die Programmvorführung bei Demos häufig in eine »wilde« Sucherei auf der Platte aus.



Abbildung 6.46:
Der Application Caller

Aber auch um die täglich anfallenden, sich oft wiederholenden Programmaufrufe und Arbeitsabläufe u.a. bei Anwendern und Programmierern zu rationalisieren sowie die umfangreiche Programmverwaltung (Programm- Batchaufrufe, Befehle) auf der Festplatte zu vereinfachen und wesentlich übersichtlicher zu gestalten, soll eine individualisierbare Programmschnellstartleiste, aus der Windows-Applikationen einheitlich aufgerufen werden können und die eine einheitliche Programmschnellstartfläche bietet, entwickelt werden (Abbildung 6.46) – ein dynamischer Application Caller.

Der Anwender Herr Maier muss jeden Abend die Lohndatei mit dem Befehl 'Copy C:\Lohn\Mitarb\Lohn.Dat A:\Lohn\Mitarb\Lohn.Dat' auf Diskette sichern!

betriebliches Anwendungsbeispiel

Es ist nicht sinnvoll und ratsam, diesen Befehl immer wieder eintippen zu müssen, weil die Fehleranfälligkeit und der Zeitaufwand (summiert sich täglich) zu groß sind.

Sie werden mir nun entgegenhalten, dass man für diese Aufgabe ein Batchprogramm, welches diese Arbeit übernimmt, kreiert oder täglich den Explorer für diese Art Aufgaben bemüht. Keine Frage, dies ist auch möglich, birgt aber den Nachteil, dass sich der Anwender bei mehreren Sicherungs- und Programmaufrufen, die Batchprogrammnamen und evtl. die Verzeichnisnamen, Befehle etc. merken muss.

Befreiung von Routineabläufen

Ziel kann es aber nicht sein, dass der Anwender durch diesen Arbeitsablauf implizit dazu veranlasst wird, sich die verschiedenen Aufrufe unnötigerweise zu merken oder sogar deshalb Aufschriebe (manuelle Tätigkeit) anzufertigen, sondern Ziel muss es sein, den Anwender von derartigen Routineabläufen zu befreien, damit er sich auf das »WESENTLICHE« konzentrieren kann.



Sehr oft ist es doch so, dass wir fast täglich hauptsächlich mit einer bestimmten Anzahl und Art von Anwendungsprogrammen arbeiten, also aus Windows heraus starten.

Stellen wir uns dazu vor, dass die Applikationen »Word 97«, »Explorer«, »Visual Basic«, »CD-Player« und der »Editor« primär von uns bei Bedarf innerhalb einer Windows-Sitzung aufgerufen werden.

Um die jeweilig gewünschte Anwendung starten zu können, müsste über den Explorer oder die Win95-Startleiste die entsprechende Programmgruppe (Ordner) evtl. Unterprogrammgruppe (Unterordner) etc. geöffnet werden.



Bei Windows98 hat Microsoft »mitgedacht« und quasi die Funktionalität eines Application Callers im Betriebssystem integriert.

Da die Aufrufe der Anwendungsprogramme standardmäßig in unterschiedlichen Programmgruppen (Ordnern), Untergruppen (Unterordnern), Startmenüs und deren Untermenüs stehen, bleibt nur die Vorgehensweise des ständigen und stetigen Öffnens von Programmgruppen (Ordnern) bzw. Menüs.

Eine traditionelle Möglichkeit bleibt noch übrig – nämlich die häufig benötigten Programmaufrufe (Verknüpfungen, Links) in einer globalen Programmgruppe (Ordner) oder einem globalen Menü zu hinterlegen. Beispielsweise legen Sie den Ordner (die Programmgruppe) »Meine Programme« an und stellen in diesen Ordner Ihre Applikationsaufrufe für Word 97, Explorer, Visual Basic etc. ein.

Da diese Möglichkeit auch keine zufrieden stellende Lösung ist, soll der Application Caller entwickelt werden, der uns, einmal aufgerufen, eine Programmschnellstartleiste für unsere individuellen Programmaufrufe anbietet, aus der die jeweiligen Applikationen mit einem Mausklick oder über Tastaturkürzel gestartet werden können.

Der Application Caller könnte dann somit später in die Windows-Autostartgruppe eingebunden werden und würde so, beim Windowsboot, Ihr individuelles Programmaufrufenmenü erzeugen, aus dem Sie all Ihre häufig benötigten Applikationen bequem aufrufen können.

Benutzeroberfläche

Die folgenden Leistungsmerkmale, Leistungsbeschreibungen und Restriktionen soll unser Application Caller beinhalten:

Leistungsmerkmale, Leistungsbeschreibungen und Restriktionen

- ▶ Einfach zu handhabendes Generier- und Lösungskonzept.
- ▶ Der Aufbau von individuellen Programmschnellstartleisten, bezogen auf das jeweilige Aufgabengebiet, soll ermöglicht werden.
- ▶ Über einen beliebigen ASCII/ANSI-Text-Editor (z.B. Windows Notepad) soll der Leistenaufbau (Applikations-Aufrufe für die Programmschnellstartleiste) als Text variabel hinterlegt werden können.
- ▶ Maximal 15 Zeilen (15 Programmaufrufe) sollen generierbar sein.
- ▶ Anwendungen sollen durch Setzen generatorspezifischer Kennzeichen wahlweise im Vollbild, Normalbild oder als Symbol gestartet werden können.
- ▶ Rationalisierung von Routineabläufen und dadurch Zeitersparnis.
- ▶ Einheitliche, übersichtliche und einfach zu bedienende Benutzeroberfläche.
- ▶ Die Programmschnellstartleiste soll frei auf dem Bildschirm positioniert werden können und soll auch diese Position beibehalten, wenn der Application Caller beendet und erneut aufgerufen wird. Die Bildschirmkoordinaten müssen somit beim Verlassen des Application Callers gemerkt werden. Dadurch entfällt das oft lästige Verschieben eines Anwendungsfensters an die gewünschte Bildschirmposition nach erneutem Aufruf. Die Programmschnellstartleiste des Application Callers ist nicht bildschirmmittig zu senden, sondern ist an der gleichen Stelle auf dem Bildschirm wie zuvor anzuzeigen.
- ▶ Die ausgelöste Transaktion (der Applikationsaufruf) muss asynchron gestartet werden, das heißt, die aufgerufene Anwendung wird parallel und unabhängig vom Application Caller aufgerufen, so dass sofort im Application Caller weitergearbeitet werden kann, sprich weitere Transaktionen ausgelöst werden können.

Mit diesem Utility wären wir dann in der Lage, unsere täglich anfallenden Aufgaben-/Arbeitsabläufe aus einer Benutzeroberfläche heraus zu koordinieren, zu steuern und zu starten.

Lösung

Damit der Application Caller seine Arbeit aufnehmen kann, erstellen oder modifizieren wir mit einem beliebigen Editor (z.B. Notepad), der im ASCII/ANSI-Format speichert, folgende Datei:

6 Workshop: Mit den Steuerelementen programmieren

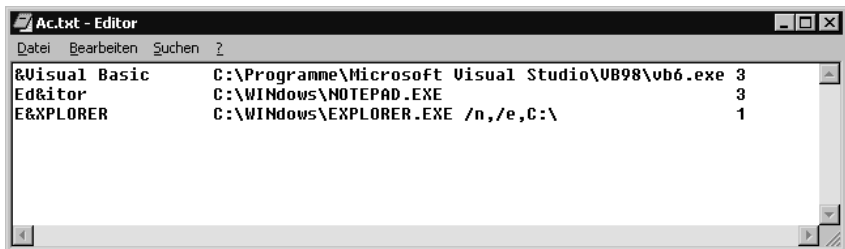
Startleistendatei AC.TXT Der Application Caller liest diese Startleistendatei (Abbildung 6.47) und erhält Informationen darüber, welche Programmaufruftexte (Menütexte) innerhalb der Benutzeroberfläche generiert bzw. gesendet werden müssen und durch welchen Programmaufruf das aus der Startleiste ausgewählte Programm in welchem WindowStyle (Normal-, Vollbild oder Symbol) gestartet werden soll.

Das bedeutet, dass die zu sendenden Texte für den Dialog von Ihnen individuell über diese Datei gestaltet werden können. Damit ist eine vollkommene Variabilität gewährleistet.

Konvention mit Beispiel ab Zeile 1 Spalte 1 beginnend :

Konvention in Zeile 1 Spalte 1 beginnend
Spalte 001 bis 017 frei wählbarer Programmaufrufstext
Spalte 020 bis 064 Applikationsaufruf
Spalte 070 Kennzeichen für WindowStyle(Voll-, Normalbild oder Symbol)

Abbildung 6.47:
Beispiel einer
möglichen Startleis-
tendatei AC.TXT

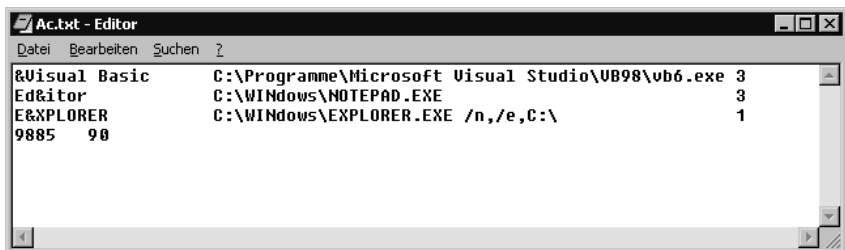


```
Ac.txt - Editor
Datei Bearbeiten Suchen ?
&Visual Basic C:\Programme\Microsoft Visual Studio\VB98\vb6.exe 3
Ed&itor C:\WINDOWS\notepad.exe 3
E&XPLORER C:\WINDOWS\EXPLORER.EXE /n,/e,C:\ 1
```

Bildschirm- koordinaten

Die Startleistendatei AC.TXT (Abbildung 6.47) darf max. 15 Zeilen (Einträge für Applikationsaufrufe) enthalten, das heißt der Application Caller kann eine Programmschnellstartleiste mit höchstens 15 Programmaufrufen generieren.

Abbildung 6.48:
Die Bildschirmkoo-
rdinaten merkt sich
der Application
Caller



```
Ac.txt - Editor
Datei Bearbeiten Suchen ?
&Visual Basic C:\Programme\Microsoft Visual Studio\VB98\vb6.exe 3
Ed&itor C:\WINDOWS\notepad.exe 3
E&XPLORER C:\WINDOWS\EXPLORER.EXE /n,/e,C:\ 1
9885 90
```

Die letzte Zeile in Ihrer Startleistendatei (Abbildung 6.48) sollten Sie nie löschen oder verändern, da dieser Eintrag vom Application Caller selbst geschrieben und zur Positionierung der Programmschnellstartleiste auf dem Bildschirm benötigt wird. Der Eintrag lautet momentan 9885 90.

Was bedeutet nun der Eintrag:

Bedeutung

```
&Visual Basic C:\Programme\Microsoft Visual Studio\VB98\vb6.exe
3
```

Da dies der erste Eintrag in der Startleistendatei ist, wird bei Aufruf des Application Callers der Programmaufruf »&Visual Basic« ebenfalls als erste Befehlsschaltfläche der Programmschnellstartleiste generiert.

Aus dieser kann dann das Programm »Visual Basic« entweder durch Mausklick auf die Befehlsschaltfläche oder durch den Shortcut `[ALT]-[V]` (&V) über die Tastatur mit dem Programmaufruf `C:\Programme\Microsoft Visual Studio\VB98\vb6.exe` gestartet werden.

In welchem Modus die jeweilige Anwendung, hier in unserem Fall »Visual Basic«, auf dem Bildschirm erscheinen soll, kann durch die numerische Kennzeichnung des WindowStyle in Stelle 70 definiert werden.

WindowStyle

Dabei gelten folgende Möglichkeiten :

Nummerischer Wert	WindowStyle
1, 5, 9	Im Normalbild, und die aufgerufene Anwendung hat den Fokus (ist aktiviert).
2	Als Symbol, und die aufgerufene Anwendung hat den Fokus (ist aktiviert).
3	Im Vollbild, und die aufgerufene Anwendung hat den Fokus (ist aktiviert).
4, 8	Im Normalbild, und die aufgerufene Anwendung hat nicht den Fokus (ist nicht aktiviert).
6, 7	Als Symbol, und die aufgerufene Anwendung hat nicht den Fokus (ist nicht aktiviert).

Tabelle 6.1:
WindowStyle

Um die Startleistendatei AC.TXT in dem Laufwerk und Pfad öffnen zu können, in dem Sie den Application Caller installiert haben, wird der Applikationspfad (*app.Path*) gebraucht, der in der Ereignis-Prozedur *Form_Load* ausgelesen wird.

**Funktionsweise
AC_1.FRM**

```
' Pfad und Dateiname in Variable stellen
aktpfad = app.Path
If Right$(aktpfad, 1) <> "\" Then
    aktpfad = aktpfad + "\"
End If
DBDat$ = aktpfad & "AC.TXT"
```

Danach können die Einträge der Startleistendatei in die im Deklarationsteil auf 15 Zeilen definierte Tabelle `CALLER$()` zur weiteren Verarbeitung gefüllt werden (Die Dimensionierung lässt sich natürlich beliebig erweitern).

```
' Callerdatei EINLESEN
Open DBDat$ For Input As dateinr
M = -1
```

```

While Not EOF(dateinr)
    M = M + 1
    ' Im Fehlerfall beenden
    If M > 15 Then End
    Line Input #dateinr, Caller$(M)
Wend
Close dateinr
    
```

Ist der Lese-Index M größer als 15, findet das Utility sein Ende.

Abbildung 6.49:
Application Caller
mit nur einem
Eintrag



Da ja der Application Caller eine ganz entscheidende Eigenschaft besitzen soll, nämlich eine Programmschnellstartleiste mit einer variablen Anzahl von Programmaufrufmöglichkeiten (Abbildungen 6.49 und 6.50) zur Laufzeit generieren zu können, muss vom »herkömmlichen« Lösungsansatz Abstand genommen werden.

Abbildung 6.50:
Application Caller
mit den in
Abbildung 6.47
definierten
Einträgen



Würde man eine fixe Anzahl Befehlsschaltflächen zur Aufnahme von Programmaufrufen in die Form einbetten, so wären, wenn weniger Programmaufrufe als Schaltflächen vorhanden, zu viele, sprich leere, Schaltflächen zu sehen. Auf der anderen Seite kann es passieren, dass die fixe Anzahl Befehlsschaltflächen in der Form nicht ausreicht, da mehr zu generierende Programmaufrufe als Schaltflächen vorhanden sind.



Stellen Sie sich vor, es existieren in der Form des Application Callers sechs Befehlsschaltflächen fix, deren Inschriften (Captions) bei Laufzeit aus der Startleistendatei *AC.TXT* mit den Programmaufrufertexten gefüllt werden.

Wären nur vier Programmaufrufe in der Startleistendatei hinterlegt, würden im Application Caller zwei Schaltflächen leer sichtbar bleiben. Wären hingegen sieben Programmaufrufe in der Startleistendatei hinterlegt, so könnte der siebte Programmaufruf gar nicht generiert werden, weil keine siebte Befehlsschaltfläche in der Form existent ist.

Eine weitere Möglichkeit wäre, da maximal 15 Programmaufrufe zu realisieren sind, genau 15 Befehlsschaltflächen im Entwicklungsmodus in die Form einzubetten und die nicht benötigten zur Laufzeit auf nicht sichtbar zu setzen (*Visible = False*).

Aber da wäre dann u. a. das Problem zu lösen, die im Entwicklungsmodus definierte Größe der Form, bedingt durch die höchste Anzahl an Befehlsschaltflächen, zur Laufzeit zu verringern etc. Und dann wäre da noch der »verschrenkte« Speicherplatz, wenn weniger Programmaufrufe als eingebettete Schaltflächen zu generieren sind usw.

Das Fazit aus dieser Problemanalyse lautet doch wohl, es muss auf irgendeine andere Art und Weise möglich sein, Befehlsschaltflächen, allgemein Objekte, dynamisch erst bei Laufzeit der Applikation anzulegen – zu erzeugen.

Dynamische Anlage von Objekten

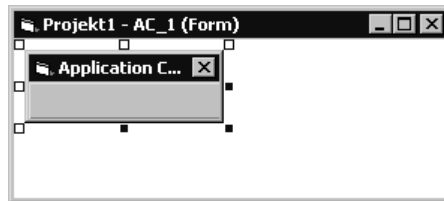


Abbildung 6.51: Formular AC_1.FRM des Application Callers

In der Form des Application Callers ist ausschließlich ein Objekt *Befehlsschaltfläche* (Abbildung 6.51), wir nennen es der Einfachheit halber »Masterobjekt Befehlsschaltfläche«, eingebettet. Alle weiteren Schaltflächen werden bei Bedarf, der abhängig von der Anzahl der Programmaufrufe aus der Startleistendatei ist, dynamisch durch Bildung von Instanzen angelegt.

Jede neu angelegte Instanz *Befehlsschaltfläche* stammt von der ersten Befehlsschaltfläche, unserem Masterobjekt, ab und erbt somit auch automatisch die Eigenschaften des Masterobjektes.

Instanzen



Abbildung 6.52: Befehlsschaltfläche indizieren

Um Objekte überhaupt dynamisch anlegen und ansprechen zu können, muss das jeweilige Objekt, in unserem Fall die Befehlsschaltfläche, indiziert werden. Dazu ist die Eigenschaft *Index* der ersten Schaltfläche in der Form (das Masterobjekt), auf 0 zu setzen (Abbildung 6.52).

Weitere Befehlsschaltflächen werden dynamisch durch Inkrement der Eigenschaft *Index* (um 1 erhöhen) erzeugt und sind auch über diese Indexnummer ansprechbar.

Vergleichbar ist diese Vorgehensweise mit einer zur Laufzeit zu dimensionierenden Tabelle, z. B.

```
DIM NAMEN$(I)
```

Ebenso wie Variablen dimensioniert und indiziert werden können, können Objekte als Steuerelementefeld dimensioniert und indiziert werden.

```
' 1. Originalbutton  
Cmd(0).Caption = Mid$(Caller$(0), 1, 17)
```

Der erste Programmaufruftext aus der Tabelle *Caller\$(i)* wird in die Eigenschaft *Caption* der ersten Befehlsschaltfläche (Masterobjekt), die den Namen *CMD* indiziert mit *0* trägt, gefüllt.

Danach werden in einer Schleife, die die Anzahl Programmaufrufe repräsentiert (*M*), weitere Schaltflächen durch Instanzenbildung angelegt.

```
For i = 1 To M - 1  
    Load Cmd(i) ' Neuen Button erzeugen  
    ' Setze neuen Button unter letzten Button  
    Cmd(i).Top = Cmd(i - 1).Top + 360  
    Cmd(i).Visible = True ' Neuen Button anzeigen  
    Cmd(i).Caption = Mid$(Caller$(i), 1, 17)  
    ' Formgröße anpassen  
    AC_1.Height = AC_1.Height + 360  
Next i
```

**die Programm-
schnellstartleiste
entsteht**

Eine neue Schaltfläche wird mit der Methode *Load* unter der darüber liegenden Schaltfläche durch Indizierung dynamisch erzeugt, angezeigt (Eigenschaft *Visible = True*) und mit dem Programmaufruftext aus der Startleistendatei *AC.TXT* versorgt (*Caption =.....*).

Danach muss noch die Größe der Form entsprechend angepasst werden (*AC_1.Height = AC_1.Height + 360*), damit der neue Button überhaupt im Formular sichtbar wird.

```
' Form wie zuletzt positionieren  
AC_1.Left = Val(Trim$(Mid$(Caller$(M), 1, 6)))  
AC_1.Top = Val(Trim$(Mid$(Caller$(M), 8, 6)))
```

Damit die generierte Programmschnellstartleiste nicht irgendwo auf dem Bildschirm erscheint, werden die Bildschirmkoordinaten von der letzten Application *Caller* Sitzung, die automatisch in der Startleistendatei *AC.TXT* gemerkt wurden, zur Positionierung verwendet.


```
Sub Cmd_Click (Index As Integer)
    X = Shell(LTrim$(RTrim$(Mid$(Caller$(Index), 20, 45))),
Val(Mid$(Caller$(Index), 70, 1)))
End Sub
```

Ebenfalls interessant an dieser objektindizierten Vorgehensweise ist, dass es nicht für »n« Schaltflächen »n« Ereignisprozeduren gibt, sondern ausschließlich eine, in der allerdings auf alle Schaltflächen einzeln über den Index Bezug genommen werden kann.

Durch Mausklick auf eine Befehlsschaltfläche der Programmschnellstartleiste oder Aktivierung über Tastenkürzel wird über die Funktion *SHELL* der entsprechende Applikationsaufruf, der in der Tabelle *Caller\$()* steht, realisiert (z.B. C:\Programme\Microsoft Visual Studio\VB98\vb6.exe).

```
' Callerdatei zurückschreiben, um die Formposition zu speichern
Dim dateinr As Integer
dateinr = FreeFile
' Formposition speichern
Caller$(M) = Space$(15)
Mid$(Caller$(M), 1, 6) = AC_1.Left
Mid$(Caller$(M), 8, 6) = AC_1.Top
Open DBDat$ For Output As dateinr
For i = 0 To M
    Print #dateinr, Caller$(i)
Next i
Close dateinr
End
```

FORM_UNLOAD

Damit die Programmschnellstartleiste bei erneutem Aufruf des Application Callers genau an der gleichen Stelle auf dem Bildschirm erscheint, an der sie jetzt momentan beim Beenden des Application Callers steht, werden die Bildschirmkoordinaten der Form als letzter Eintrag in die Startleistendatei *AC.TXT* geschrieben.

Bevor Sie den Application Caller erfolgreich starten können, müssen Sie entsprechend Ihren Anforderungen und Wünschen Ihre individuellen Aufrufe (Programmaufruftexte, Programmaufrufe und Kennzeichen für WindowStyle) in der Startleistendatei *AC.TXT* (Abbildung 6.53) mit einem Windows-Editor, z. B. Notepad, hinterlegen.

Application Caller aufrufen

Wird diese Startleistendatei gespeichert und der Application Caller danach gestartet, so generiert dieser daraus die folgende Programmschnellstartleiste (Abbildung 6.54).

Passen Sie Ihre individuelle Startleistendatei *AC.TXT* durch Ändern, Löschen oder Hinzufügen von Programmaufrufen an, so generiert der Application Caller bei Start daraus erneut die entsprechende Programmschnellstartleiste.

Abbildung 6.53:
Auslieferungszu-
stand der Startleis-
tendatei AC.TXT

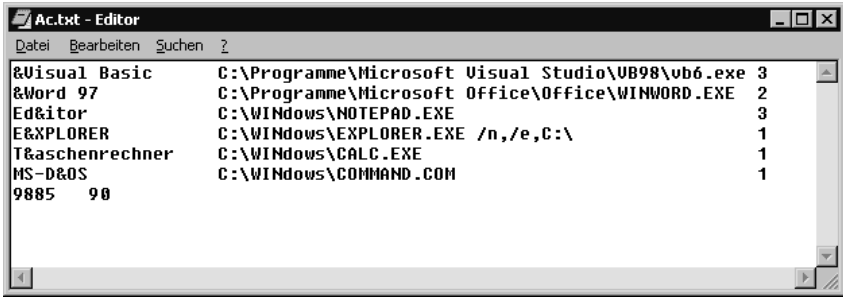


Abbildung 6.54:
Generierte
Programmschnell-
startleiste aus dem
Auslieferungszu-
stand der Startleis-
tendatei AC.TXT



6.6 Das Steuerelement »Frame«

Der Rahmen (Frame) dient der Aufnahme mehrerer Objekte, die funktionell gruppiert werden sollen. Auch hier kann über Eigenschaften sein Aussehen verändert werden.



6.6.1 Übung: Auf Anforderung einen Rahmenstil, Rahmentext setzen

Über eine Befehlsschaltfläche (CommandButton) soll die Bezeichnung des Rahmens auf »Auswahlrahmen« gesetzt werden (Abbildung 6.55).

Abbildung 6.55:
Darstellung ohne
Rahmen



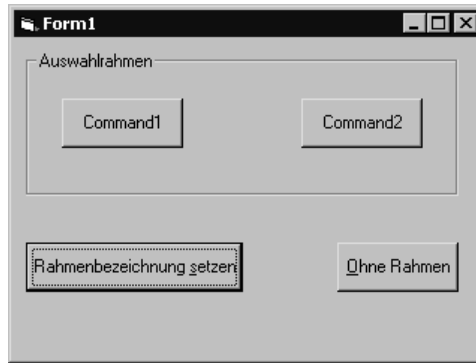


Abbildung 6.56:
Rahmenbezeichnung
setzen

Über eine zweite Befehlsschaltfläche soll es möglich werden, den Rahmen zu eliminieren (Abbildung 6.56).

Lösung

Damit bei Mausklick auf den Button »Rahmenbezeichnung setzen« auch der gewünschte Rahmenbezeichnungstext im Frame erscheint, ist die Ereignisprozedur *Click* des *CommandButtons* wie folgt zu erweitern.

```
Private Sub Command3_Click()
    Frame1.Caption = "Auswahlrahmen"
End Sub
```

Dem Rahmen *Frame1* ist die auszugebende Bezeichnung (*Auswahlrahmen*) in der Eigenschaft *Caption* (Inschrift, Überschrift) zu übergeben.

Starten Sie nun Ihre Applikation z. B. über **[F5]** und aktivieren den Button *Rahmenbezeichnung setzen*, so wird, wie in Abbildung 6.55 ersichtlich, die Bezeichnung des Rahmens ausgegeben.

Damit bei Mausklick auf den Button *Ohne Rahmen* der Rahmen im Formular nicht mehr angezeigt wird, ist die Ereignisprozedur *Click* des *CommandButtons* anzupassen.

```
Private Sub Command4_Click()
    Frame1.BorderStyle = 0
End Sub
```

Über die Eigenschaft *BoderStyle* (Rahmenstil) können Sie bestimmen, ob ein fester Rahmen (*BorderStyle* = 1) oder kein Rahmen (*BorderStyle* = 0) zur Anzeige kommen soll.

Aktivieren Sie nach dem Starten Ihrer Applikation den Button *Ohne Rahmen*, so wird, wie in Abbildung 6.56 zu sehen, die Button-Gruppe ohne umschließenden Rahmen angezeigt.

6.7 Das Steuerelement »DriveListBox«

Mit dem Laufwerkslistenfeld kann der Benutzer verfügbare Laufwerke auswählen und zwischen ihnen wechseln. Auf sein Aussehen hat man keinen Einfluss.



6.7.1 Übung: Einen Laufwerkswechsel anzeigen

Das Laufwerkslistenfeld soll bei Applikationsstart standardmäßig auf die Festplatte C: verweisen. Ein Laufwerkswechsel ist im Steuerelement *Label* (Bezeichnungsfeld) unter Angabe des Laufwerksbuchstaben zu visualisieren (Abbildung 6.57).

Abbildung 6.57:
Mit dem gewählten
Laufwerksbuch-
staben geladenes
Bezeichnungsfeld



Lösung

Damit bei Applikationsstart auf das Laufwerk C: verwiesen wird, ist die Ereignisprozedur *Load* des Formulars zu erweitern.

```
Private Sub Form_Load()  
    Drive1.Drive = "C:"  
    Label2.Caption = "C:"  
End Sub
```

Form_Load ist die Prozedur, die noch vor dem eigentlichen Anzeigen des Formulars auf dem Bildschirm aufgerufen wird. In der Eigenschaft *Drive* des Laufwerkslistenfeldes, die zur Entwurfszeit im Eigenschaftfenster nicht verfügbar ist, legen Sie den Laufwerksbuchstaben gefolgt von einem Doppelpunkt fest und geben diesen zur Visualisierung im Bezeichnungsfeld (*Label*) aus.



Über die Eigenschaft *Drive* können alle im System vorhandenen oder mit dem System verbundenen Laufwerke, die gültig sind, angesprochen werden, und zwar zu dem Zeitpunkt, an dem das Steuerelement erstellt oder zur Laufzeit aktualisiert wurde.

Dem Bezeichnungsfeld *Label2* ist das im Laufwerkslistenfeld *Drive1* ausgewählte Laufwerk als Laufwerksbuchstabe in der Eigenschaft *Caption* zu übergeben. Dazu ist die Ereignisprozedur *Change* des Laufwerkslistenfeldes zu erweitern.

```
Private Sub Drive1_Change()
    Label2.Caption = Drive1.Drive
End Sub
```

Das Event *Change* wird immer dann aufgerufen, wenn der Benutzer ein anderes Laufwerk auswählt. Die Eigenschaft *Drive* enthält dabei den ausgewählten Laufwerksbuchstaben. Daraus folgt, dass *Drive* das ausgewählte Laufwerk zurückgibt oder in *Drive* ein bestimmtes Laufwerk zur Anzeige gesetzt werden kann.

Starten Sie nun Ihre Applikation und wechseln das Laufwerk, so erscheint, wie in Abbildung 6.57 zu sehen, das ausgewählte Laufwerk in Form des Laufwerksbuchstaben im Bezeichnungsfeld (*Label*).

6.8 Das Steuerelement »DirListBox«

Mit dem Verzeichnislistenfeld kann der Benutzer verfügbare Verzeichnisse (Ordner) innerhalb eines Laufwerks auswählen und zwischen ihnen wechseln. Auf sein Aussehen hat man keinen Einfluss.

6.8.1 Übung: Ordner eines Laufwerks listen

Das Verzeichnislistenfeld soll bei Applikationsstart standardmäßig die Ordnerstruktur des Hauptverzeichnisses (Rootverzeichnis) der Festplatte C: anzeigen (Abbildung 6.58).

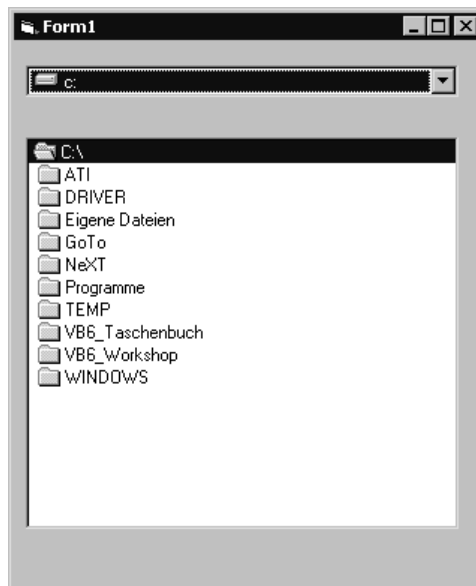
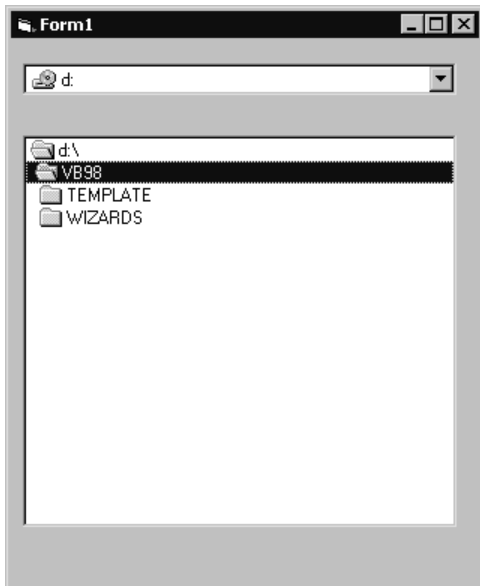


Abbildung 6.58: Ordnerstruktur des Hauptverzeichnisses (Rootverzeichnis) der Festplatte C:

Abbildung 6.59:
Ordnerstruktur
eines anderen
gewählten
Laufwerks



Bei einem Laufwerkswechsel ist die jeweilige Ordnerstruktur des gewählten Laufwerks zu visualisieren (Abbildung 6.59).

Lösung

Damit bei Applikationsstart auf das Rootverzeichnis der Festplatte C: verwiesen wird, ist die Ereignisprozedur *Load* des Formulars zu erweitern.

```
Private Sub Form_Load()  
    Drive1.Drive = "C:"  
    Dir1.Path = "C:\"  
End Sub
```

In der Eigenschaft *Drive* des Laufwerkslistenfeldes legen Sie das Laufwerk fest, mit dem standardmäßig gestartet werden soll. Der Eigenschaft *Path* des Verzeichnislistenfeldes ist der Pfad, der angezeigt werden soll, komplett inklusive Laufwerksbuchstaben zu übergeben.

Wenn Sie Ihre Applikation jetzt starten, so wird die Ordnerstruktur des Hauptverzeichnisses (Rootverzeichnis) der Festplatte C:, wie in Abbildung 6.58 ersichtlich, angezeigt.

Um die Ordnerstruktur eines neu angewählten Laufwerks anzeigen zu können, muss dem Verzeichnislistenfeld *Dir1* das im Laufwerkslistenfeld *Drive1* ausgewählte Laufwerk in der Eigenschaft *Path* übergeben werden. Dazu ist die Ereignisprozedur *Change* des Laufwerkslistenfeldes wie folgt zu erweitern:

```
Private Sub Drive1_Change()  
    Dir1.Path = Drive1.Drive  
End Sub
```

Das Event *Change* wird immer dann aufgerufen, wenn der Benutzer ein anderes Laufwerk auswählt. Die Eigenschaft *Drive* enthält dabei den ausgewählten Laufwerksbuchstaben.

Starten Sie nun Ihre Applikation und wechseln das Laufwerk, so erscheint, wie in Abbildung 6.59 zu sehen, die Ordnerstruktur des ausgewählten Laufwerks. Es ist Ihnen nun mit diesen beiden Steuerelementen möglich, sich vollständig durch Ihre existenten Laufwerke und die darin enthaltenen Ordner zu bewegen.

Was jetzt noch fehlt, ist eine Anzeige der Dateien innerhalb eines Ordners. Im nächsten Kapitel erfahren Sie, wie mit einem weiteren Steuerelement auch diese Aufgabe gelöst wird.

6.9 Das Steuerelement »FileListBox«

Mit dem Dateilistenfeld kann der Benutzer alle verfügbaren Dateien eines Verzeichnisses (Ordners) innerhalb eines Laufwerks anzeigen lassen und daraus einzelne z.B. zur Weiterverarbeitung etc. auswählen. Auf sein Aussehen hat man keinen Einfluss.

6.9.1 Übung: Dateien eines Ordners listen



Das Dateilistenfeld soll bei Applikationsstart standardmäßig alle Dateien des Hauptverzeichnisses (Rootverzeichnis) der Festplatte C: anzeigen (Abbildung 6.60).

Bei einem Laufwerks- oder Verzeichniswechsel sind wiederum alle Dateien des gewählten Ordners innerhalb des gewählten Laufwerks zu visualisieren (Abbildung 6.61).

Lösung

Damit bei Applikationsstart auf das Rootverzeichnis der Festplatte C: verwiesen wird, ist die Ereignisprozedur *Load* des Formulars zu erweitern.

```
Private Sub Form_Load()  
    Drive1.Drive = "C:"  
    Dir1.Path = "C:"  
End Sub
```

Form_Load ist die Prozedur, die noch vor dem eigentlichen Anzeigen des Formulars auf dem Bildschirm aufgerufen wird. In der Eigenschaft *Drive* des Laufwerkslistenfeldes, die zur Entwurfszeit im Eigenschaftenfenster nicht verfügbar ist, legen Sie den Laufwerksbuchstaben gefolgt von einem Doppelpunkt fest.

Abbildung 6.60:
Alle Dateien des
Hauptverzeichnisses
der Festplatte
C:

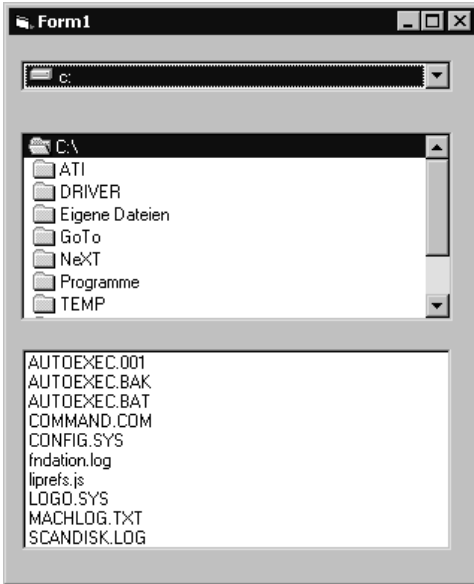
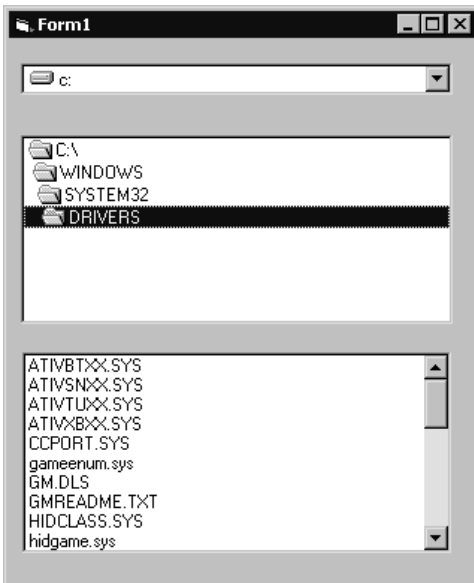


Abbildung 6.61:
Alle Dateien
eines anderen
gewählten Ordners



Der Eigenschaft *Path* des Verzeichnislistenfeldes ist der Pfad, der angezeigt werden soll, komplett inklusive Laufwerksbuchstaben zu übergeben.

Um die Ordnerstruktur eines neu angewählten Laufwerks anzeigen zu können, muss dem Verzeichnislistenfeld *Dir1* das im Laufwerkslistenfeld *Drive1* ausgewählte Laufwerk in der Eigenschaft *Path* übergeben werden. Dazu ist die Ereignisprozedur *Change* des Laufwerkslistenfeldes wie folgt zu erweitern:


```
Private Sub Drive1_Change()
    Dir1.Path = Drive1.Drive
End Sub
```

Wird ein anderer Ordner ausgewählt, um sich die darin befindlichen Dateien anzeigen zu lassen, so ist dazu die Ereignisprozedur *Change* des Verzeichnislistenfeldes anzupassen.

```
Private Sub Dir1_Change()
    File1.Path = Dir1.Path
End Sub
```

Dem Dateilistenfeld *File1* ist in der Eigenschaft *Path* der im Verzeichnislistenfeld *Dir1* ausgewählte Ordner zu übergeben. Das Event *Change* wird immer dann aufgerufen, wenn der Benutzer ein anderes Verzeichnis wählt.

Starten Sie nun Ihre Applikation und wechseln den Ordner, so erscheinen, wie in Abbildung 6.61 zu sehen, alle Dateien des ausgewählten Ordners innerhalb des aktiven Laufwerks.

Es ist Ihnen nun mit diesen drei Steuerelementen möglich, sich durch Ihre existenten Laufwerke, die darin enthaltenen Ordner und die wiederum darin enthaltenen Dateien vollständig zu bewegen.

6.10 Das Steuerelement »Label«

Im Bezeichnungsfeld (Label) kann ein Text angezeigt werden, der normalerweise nicht während des Programmlaufs vom Benutzer geändert werden kann. Sein Aussehen ist durch Eigenschaften beeinflussbar.

6.10.1 Übung: Text laden und zentrieren

Über eine Befehlsschaltfläche (*CommandButton*) soll der Text »Visual Basic 6.0 ist SUPER!« in einem Bezeichnungsfeld (*Label*) erscheinen (Abbildung 6.62).

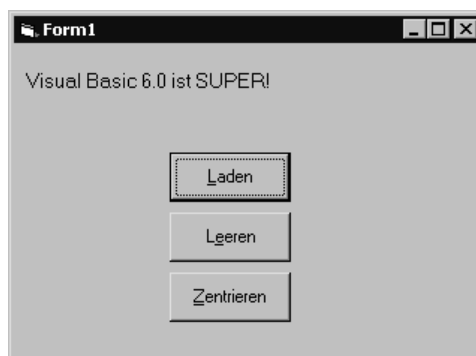


Abbildung 6.62:
Text ins Bezeichnungsfeld laden

Abbildung 6.63:
Geladenen Text im
Bezeichnungsfeld
zentrieren



Über eine weitere soll es möglich werden, diesen Text innerhalb des Bezeichnungsfeldes mittig darzustellen, zu zentrieren (Abbildung 6.62). Mit einer dritten Befehlsschaltfläche soll der Inhalt des Bezeichnungsfeldes geleert (initialisiert) werden können.

Lösung

Damit bei Mausklick auf den Button »Laden« auch der gewünschte Text im Label erscheint, ist die Ereignisprozedur *Click* des ersten CommandButtons zu erweitern.

```
Private Sub Command1_Click()  
    Label1.Caption = "Visual Basic 6.0 ist SUPER!"  
End Sub
```

Dem Bezeichnungsfeld *Label1* ist der auszugebende Text (*Visual Basic 6.0 ist SUPER!*) in der Eigenschaft *Caption* (Inschrift, Überschrift) zu übergeben.

Beliebiger Text kann in einem Bezeichnungsfeld (*Label*) über das Setzen der Eigenschaft *Alignment* ausgerichtet werden. Dabei gelten folgende unterschiedliche Einstellmöglichkeiten:

- ▶ *Alignment* = 0 Text wird links ausgerichtet
- ▶ *Alignment* = 1 Text wird rechts ausgerichtet
- ▶ *Alignment* = 2 Text wird zentriert

```
Private Sub Command2_Click()  
    Label1.Alignment = 2  
End Sub
```

So ist nur die Ereignisprozedur *Click* des entsprechenden CommandButtons um das Setzen des gewünschten *Alignment*-Werts zu modifizieren.

Damit beim Mausklick auf den dritten Button »Leeren« kein Text im Label erscheint, ist die Ereignisprozedur *Click* des dritten CommandButtons zu erweitern.

```
Private Sub Command3_Click()
    Label1.Caption = ""
End Sub
```

Der Eigenschaft *Caption* des Labels wird ein Leerstring, sprich nichts, übergeben.

Starten Sie nun Ihre Applikation z.B. über **[F5]** und aktivieren den Button *Laden*, so erscheint, wie in Abbildung 6.62 zu sehen, der gewünschte Text im Bezeichnungsfeld (Label). Aktivieren Sie hingegen den Button *Leeren*, enthält Ihr Label-Objekt nichts.

6.10.2 Übung: Benutzeroberfläche zur Laufzeit anpassen

Der Wunsch der Softwareanwender nach individuell anpassbaren Benutzeroberflächen oder zumindest Einzelformularen innerhalb einer großen Applikation, z. B. betriebswirtschaftlicher Software, wird immer lauter.



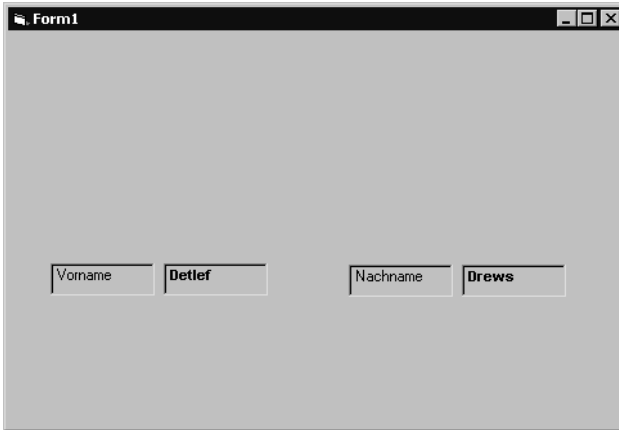
Abbildung 6.64:
Startformular im
Originalzustand

So soll es, wie in den Abbildungen 6.64 und 6.65 zu sehen, möglich werden, Adressausgabefelder (*Label*) z. B. innerhalb einer Adressverwaltung zur Laufzeit beliebig auf dem Bildschirm anordnen zu können.

Lösung

Damit wir merken, welche Felder (Vorname oder Nachname) der Anwender auf dem Bildschirm für das freie Verschieben durch Mausklick ausgewählt hat, definieren wir zwei globale numerische Variablen (*vn* für Vorname und *nn* für Nachname) und initialisieren diese bei Applikationsstart mit dem Wert 0.

Abbildung 6.65:
Zur Laufzeit
angepasste
Benutzeroberfläche



```
Option Explicit
Dim vn As Integer
Dim nn As Integer
Private Sub Form_Load()
    vn = 0
    nn = 0
End Sub
```

Wird auf das Bezeichnungsfeld mit der Inschrift (Caption) *Vorname* geklickt, werden also die beiden Label *Vorname* und *Detlef* für das freie Verschieben ausgewählt, so sind die Variablen *vn* und *nn* entsprechend zu setzen (*vn* auf 1 und *nn* auf 0).

```
Private Sub Label1_Click()
    If vn = 0 Then
        vn = 1
    Else
        vn = 0
    End If
    nn = 0
End Sub
```

Wählt der Anwender durch Mausklick den Nachnamen (Bezeichnungsfeld *Label2*) zum freien Anordnen, so sind wiederum die Variablen *vn* und *nn* entsprechend zu setzen (*nn* auf 1 und *vn* auf 0).

```
Private Sub Label2_Click()
    If nn = 0 Then
        nn = 1
    Else
        nn = 0
    End If
    vn = 0
End Sub
```

Nachdem der Anwender entweder die Bezeichnungsfelder Vorname mit den Label *Vorname* und *Detlef* oder Nachname mit den Label *Nachname* und *Drews* durch Mausklick ausgewählt hat, muss als Nächstes für das freie Verschieben innerhalb des Formulars gesorgt werden.

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If vn = 1 Then
        Label1.Move X, Y
        Label3.Move X + 1800 - 480, Y
    End If
    If nn = 1 Then
        Label2.Move X, Y
        Label4.Move X + 1800 - 480, Y
    End If
End Sub
```

Beim Bewegen des Mauszeigers über dem Formular wird grundsätzlich das Ereignis *Form_MouseMove* aktiviert. So ist ein Verschieben beliebiger Objekte (Steuerelemente), hier in unserer Übung Bezeichnungsfelder, in diese Prozedur zu legen.

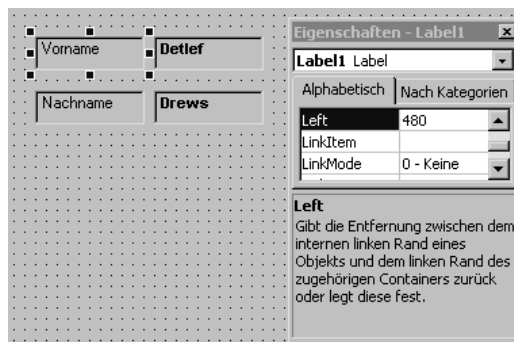


Abbildung 6.66: Eigenschaft *Left* des Vorgängers beachten und einbeziehen

Je nachdem ob Vor- oder Nachname gewählt wurde, $vn = 1$ oder $nn = 1$, kann mit der Methode *Move* das Bezeichnungsfelder-Paar *Label1* und *Label3* (Vorname und Detlef) oder *Label2* und *Label4* (Nachname und Drews) frei auf dem Formular positioniert werden.

Damit die Label *Vorname* und *Detlef* oder *Nachname* und *Drews* beim Verschieben den gleichen Abstand voneinander beibehalten, enthält die Anweisung

```
Move X + 1800 - 480, Y
```

die Parameter 1800 und 480. Wie in den Abbildungen 6.66 und 6.67 zu sehen ist, sind dies die Originalpositionen des Nachfolgers bzw. Vorgängers jeweils vom linken Rand (Eigenschaft *Left*) aus gesehen, die bei der Neupositionierung miteinzurechnen sind.

Abbildung 6.67:
Eigenschaft Left
des Nachfolgers
beachten und
einbeziehen



6.10.3 Übung: Entwicklung eines Form-Ausgabeinterpreters

Die »umfangreiche« Ausgabemaske einer in Visual Basic programmierten Adressen-Verwaltung ist für einen Neukunden anzupassen (Abbildung 6.68).

Abbildung 6.68:
Standardbild
Adress-Ausgabe

Name	<input type="text" value="Schlenker"/>
Vorname	<input type="text" value="Florian"/>
Strasse	<input type="text" value="Schülerweg 99"/>
PLZ	<input type="text" value="81250"/>
Ort	<input type="text" value="München"/>
Telefon	<input type="text" value="08010 13234"/>
Firma	<input type="text" value="Staubsauger AG"/>
Ort	<input type="text" value="Augsburg"/>
Abteilung	<input type="text" value="Verkauf"/>
Titel	<input type="text" value="Vertriebs-Ing."/>
Monatsgehalt	<input type="text" value="5450"/>
Urlaubsgeld	<input type="text" value="2450"/>
Weihnachtsgeld	<input type="text" value="3000"/>
Urlaubstage	<input type="text" value="27"/>
Hobbies	<input type="text" value="Skifahren, Tennis"/>

Fiktive Anforderung

Der Standard-Datensatz-Aufbau kann ohne Einschränkung verwendet werden (Datenbasis bleibt bestehen). Die Ausgabemaske ist wie folgt zu ändern (Abbildung 6.69):

Da die zu sendende Form mit ihren Ausgabeobjekten innerhalb des Programms »hart« kodiert wurde (im Entwicklungsmodus wurde die Anzahl Ausgabeobjekte (Bezeichnungsfelder) auch mit den Beschriftungen der Label statisch festgelegt), bleibt keine andere Wahl, als die Form entsprechend zu modifizieren und einen neuen kundenindividuellen Programmstand zu erzeugen.

Firma	Staubsauger AG
Ort	Augsburg
Abteilung	Verkauf
Name	Schlenker
Vorname	Florian
Strasse	Schülerweg 99
PLZ	81250
Ort	München
Telefon	08010 13234
Titel	Vertriebs-Ing.
Monatsgehalt	5450
Urlaubsgeld	2450
Weihnachtsgeld	3000
Urlaubstage	27
Hobbies	Skifahren, Tennis

Abbildung 6.69:
Angepasste Adress-
Ausgabe

Der zwangsläufig entstehende Programmieraufwand (kostenintensiv) summiert sich durch die Anforderungshäufigkeit.

Fazit

Aus diesem Grund soll ein Visual Basic-Modul entwickelt werden, mit dem wir dann in der Lage sind, die Bildschirmausgabefelder einer Form, die mit einem beliebigen Texteditor definiert werden können, erst bei Laufzeit der Visual Basic-Applikation zu interpretieren und anzuzeigen.

Dieses Modul soll außerdem so variabel und dynamisch sein, dass es in jede Visual Basic-Anwendung eingebunden und über die Anweisung *CALL* mit Parameterübergabe aufgerufen werden kann.

- ▶ Einfach zu handhabendes Interpreter- und Lösungskonzept.
- ▶ Über einen beliebigen Windows-Editor, z.B. Notepad, sollen die auszugebenden Felder (Feldinhalte und Feldbezeichnungen) einer Ausgabemaske (Form) als Text (Definitionsquelle), ohne Visual Basic Anweisungen, hinterlegt werden können.
- ▶ Zusätzlich sollen in der Definitionsquelle die Feldlänge und der Typ der auszugebenden Datenfelder deklariert werden können, damit bei Sendeanforderung der Form die Größe der zu generierenden Ausgabeobjekte (Label) dynamisch bestimmt und Dateninhalte von numerisch definierten Ausgabeobjekten rechtsbündig angezeigt werden können.
- ▶ Welche in der Definitionsquelle hinterlegten Felder nun tatsächlich in der zu sendenden Form angezeigt werden, sind in einer weiteren Textdatei, der Interpretersource, die ebenfalls ganz einfach über einen Windows-Editor, z.B. Notepad, erstellt werden kann, zu bestimmen.

Leistungsmerkmale, Leistungsbeschreibungen und Restriktionen

- ▶ Zusätzlich soll in der Interpretersource bestimmt werden, an welcher Position innerhalb einer Zeile die zu generierenden Ausgabeobjekte für Feldbeschreibung und Feldinhalt stehen sollen.
- ▶ Dieses Modul soll in dieser Version ausschließlich bis zu 15 Ausgabefelder interpretieren, soll aber durch entsprechende Dimensionierung im Modul verändert werden können.
- ▶ Zeitgewinn bei der Programmentwicklung und -anpassung.
- ▶ Die Ausgabemaske ist vom Programm physisch getrennt.
- ▶ Bei Bildänderungen wird es daher nicht nötig, die Applikation zu modifizieren, sondern ausschließlich die entsprechende Definition- Interpretersource.
- ▶ Entwicklungs-, Compile- und Linkläufe entfallen.
- ▶ Kundenindividuelle Wünsche über Feldplatzierungen innerhalb einer Form führen nicht zwangsläufig zu kundenindividuellen Programmständen.
- ▶ Minimierung des Programmieraufwands.
- ▶ Aufbauinterpretation erst bei Sendeanforderung der Form.
- ▶ Integration in Anwendungen jeder Art durch das Einbinden als Modul.
- ▶ Einfache Ansteuerung.
- ▶ Da die Feldbezeichnungen wie z.B. Name, Ort etc. nicht in der Form eines Visual Basic-Programms »hart« hinterlegt sind, sondern vom Programm ausgelagert in einer ASCII/ANSI-Datei stehen, lässt sich die Visual Basic-Applikation ebenfalls sehr leicht internationalisieren.
- ▶ Die Visual Basic-Anwendung kann somit ohne großen Aufwand mehrsprachenfähig ausgeliefert werden, ohne dabei das Programm modifizieren zu müssen.
- ▶ Nicht die Applikation, sondern ausschließlich die Definitionsource muss mit Hilfe eines Übersetzungstools in die gewünschte Landessprache übersetzt werden. Das Visual Basic-Programm muss nicht angepasst werden.

- Ablauflogo**
1. Definitionsource, in der alle Ausgabefelder global definiert werden, mit einem ASCII/ANSI-Texteditor, z. B. Notepad, erstellen.
 2. Interpretersource, in der die tatsächlich zu sendenden Ausgabefelder definiert werden (eine Teilmenge aus der Menge, die in der Definitionsource hinterlegt wurde), mit einem ASCII/ANSI-Texteditor erstellen.
 3. Formmodul in Ihre Visual Basic-Anwendung einbinden.
 4. Basicmodul in Ihre Visual Basic-Anwendung einbinden.
 5. Parameter übergeben und Form-Ausgabeinterpreter aufrufen.

- Einbinden**
6. Definition- und Interpretersourcedateien werden geladen, interpretiert und die Form entsprechend aufgebaut.

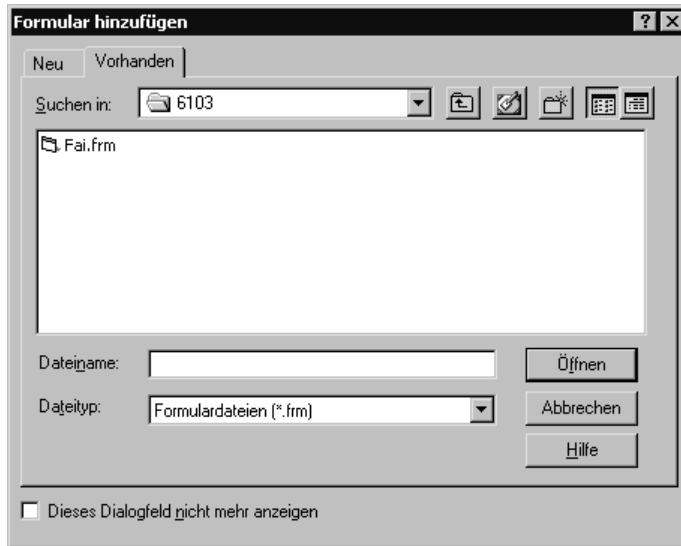


Abbildung 6.70:
Formular FAI
hinzufügen

Diese Module binden Sie ein, indem Sie Visual Basic mit Ihrer Applikation laden und danach über das Menü PROJEKT die Module *FAI.FRM* (Abbildung 6.70) und *FAI.BAS* in Ihr Programmsystem integrieren.

Das Projekt FAI (Form-Ausgabeinterpret) ist aber auch so konzipiert worden, dass es für sich allein (autonom) schon lauffähig ist, es also nicht zwingend notwendig wird, die FAI-Module in ein anderes Visual Basic-Projekt einzubinden.

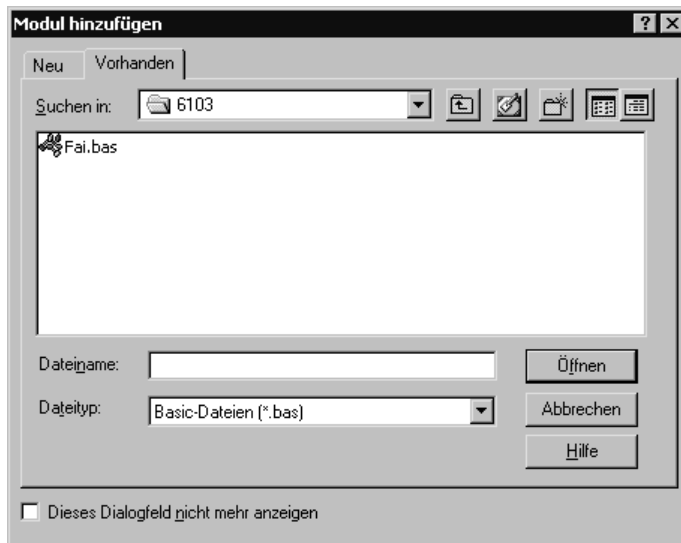


Abbildung 6.71:
BAS-Modul FAI
hinzufügen

Beispiel Zur Form TEST.FRM soll das Modul FAI.BAS mit eingebunden werden :

1. Visual Basic mit TEST.FRM laden.
2. Menü-Befehl MODUL HINZUFÜGEN aus dem Menü PROJEKT auswählen.
3. Modul FAI.BAS einladen (Abbildung 6.71).
4. Programmsystem TEST.FRM bei Beendigung speichern (Projektdatei TEST.VBP wird für Ihre Applikation neu erstellt oder modifiziert). Somit kann bei erneutem Aufruf von VB TEST das Modul FAI.BAS automatisch mitgeladen werden.

Lösung

Damit der Form-Ausgabeinterpreter seine Arbeit aufnehmen kann, erstellen oder modifizieren wir mit einem beliebigen Editor (z.B. Notepad), der im ASCII/ANSI-Format speichert, folgende Dateien:

Definitionsource F11.TXT

In diesem File definieren Sie Ihre individuellen Ausgabefelder. Der Form-Ausgabeinterpreter liest die Definitionsource und erhält Informationen darüber, wie die zu generierenden Feldbezeichnungen (Feldnamen) lauten und in welcher Größe die Ausgabeobjekte (*Label*) für die Datenfeldinhalte aufzubauen sind.

In dieser Datei hinterlegen Sie alle Ausgabefelder auf Basis der Daten Ihrer Visual Basic-Anwendung.

Stellen Sie sich dazu vor, Ihr Visual Basic-Programm ist eine Adressverwaltung mit einer Datenbasis von zehn Dateifeldern. So sollten auch alle 10 Dateifelder als Ausgabefelder in die Definitionsource eingestellt werden. Welche Ausgabefelder später in der Form tatsächlich angezeigt werden, ist in der Interpreter-source zu definieren. Damit ist eine vollkommene Variabilität gewährleistet.

Konvention mit Beispiel ab Zeile 1 Spalte 1 beginnend :

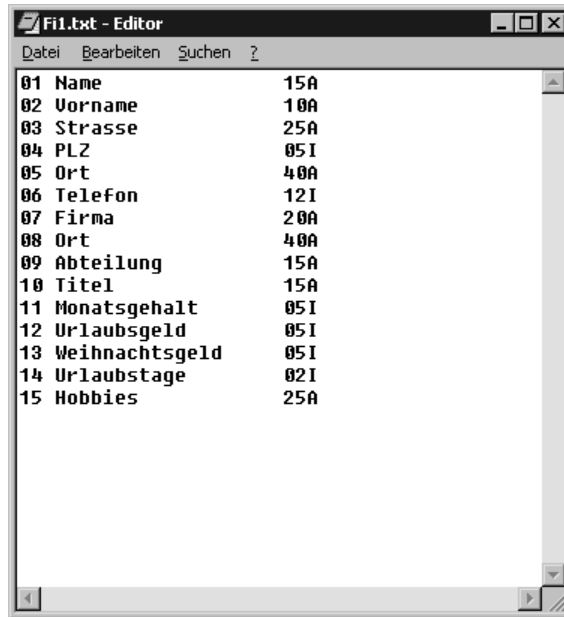
Konvention

Spalte 001 – 002, Feldnummer, die zur Referenzierung benötigt wird, von den in der Interpretersource hinterlegten Feldnummern zur Felderbasis in der Definitionsource.

Spalte 004 – 020, Feldbezeichnung (Feldname), die als Caption in ein Ausgabeobjekt (Label) einzustellen ist. Beispiel Feldbezeichnung »Vorname :«.

Spalte 023 – 024, Feldlänge des Datenfeldes, um das Ausgabeobjekt (Label) für den Datenfeldinhalt entsprechend der Datenfeldlänge dynamisch groß aufzuziehen. Beispiel : Datenfeldinhalt ist »Hans«. Das Label, um Hans auszugeben, wird nicht in der Größe 4 Zeichen (Wortlänge von Hans), sondern z.B. in der definierten Feldlänge 10 Zeichen generiert.

Spalte 025, Feldtyp. Entweder »A« = alphanummerischer Felddatentyp für linksbündige Ausrichtung des Feldinhaltes oder »I« = numerischer Felddatentyp (Ganzzahl) für rechtsbündige Ausrichtung des Feldinhaltes im Label.



Referenznummer	Feldname	Position
01	Name	15A
02	Vorname	10A
03	Strasse	25A
04	PLZ	05I
05	Ort	40A
06	Telefon	12I
07	Firma	20A
08	Ort	40A
09	Abteilung	15A
10	Titel	15A
11	Monatsgehalt	05I
12	Urlaubsgeld	05I
13	Weihnachtsgeld	05I
14	Urlaubstage	02I
15	Hobbies	25A



Abbildung 6.72:
Definitionsquelle
F11.TXT

Die Definitionsquelle *F11.TXT* darf max. 15 Zeilen (Einträge) enthalten, das heißt der Form-Ausgabeinterpreter kann höchstens 15 Ausgabefelder in einer Form generieren.

In dieser Datei hinterlegen Sie, durch Angabe der Referenznummer, ausschließlich die Felder aus der gesamten Menge aller zur Verfügung stehenden Ausgabefelder (definiert in der Definitionsquelle), die nun tatsächlich in der Form als Ausgabeobjekte (*Label*) generiert werden sollen.

Der Form-Ausgabeinterpreter liest die Interpretersource und erhält durch die Referenznummer Informationen darüber, welche Ausgabefelder aus dem gesamten Felderpool (Definitionsquelle) an welcher Position innerhalb der Form angezeigt werden sollen.

In dieser Datei können Sie somit eine Teilmenge aus der definierten Menge Ausgabefelder hinterlegen. Stellen Sie sich dazu vor, Ihr Visual Basic-Programm ist eine Adressverwaltung mit einer Datenbasis von zehn Dateifeldern, die auch alle zehn als Ausgabefelder in die Definitionsquelle eingestellt wurden.

Wie viel und welche Ausgabefelder aus diesen zehn nun wirklich in der Ausgabeform angezeigt werden, definieren Sie in der Interpretersource. Damit ist auch hier vollkommene Variabilität gewährleistet.

Konvention mit Beispiel ab Zeile 1 beginnend und weitgehendst spaltenunabhängig:

Interpretersource F10.TXT

Konvention Angabe der Feldnummer mit vorangestelltem »&«, die zur Referenzierung auf die in der Definitionsource hinterlegten Felder benötigt wird.

Praxisbeispiel 1

```
&01
      &02
      &03
    &04
    &05
    &06
      &07
      &08
    &09
    &10
    &11
&12
&13
&14
&15
```

Die Interpretersource *FIO.TXT* darf max. 15 Zeilen (Einträge) enthalten, das heißt der Form-Ausgabeinterpreter kann höchstens 15 Ausgabefelder in einer Form generieren.

Praxisbeispiel 2

```
&05
&01
      &10
      &11
          &09
          &07
```

Bedeutung Durch das Praxisbeispiel 1 werden alle Ausgabefelder, die in der Definitionsource zur Verfügung stehen, in Reihenfolge als Ausgabeobjekte versetzt in der Form generiert (Abbildung 6.73) während im Praxisbeispiel 2 deutlich wird, dass auch nur eine Teilmenge der Ausgabefelder in irgendeiner Reihenfolge als Ausgabeobjekte versetzt in der Form generiert werden können (Abbildung 6.74).

Was bedeutet nun zum Beispiel der Eintrag »&03« in der Interpretersource des Praxisbeispiels 1?

Dieser Eintrag, der als Feldnummer interpretiert wird, referenziert mit seinem Wert auf den sich hinter der Feldnummer verbergenden Inhalt der Definitionsource und besagt:

Generiere in der Form ein Ausgabeobjekt (*Label*), das die Inschrift (*Caption*) »Strasse« trägt (Feldbezeichnung, Feldname aus der Definitionsource an Feldnummer 03) und daneben ein 25 Zeichen großes Ausgabeobjekt (*Label*), welches den übergebenen Datenfeldinhalt linksbündig (definierter Feldtyp ist »A«) anzeigt.

Abbildung 6.73:
Interpretiertes und
generiertes Praxis-
beispiel 1

Abbildung 6.74:
Interpretiertes und
generiertes Praxis-
beispiel 2

Der Eintrag »&11« in der Interpretersource des Praxisbeispiels 2 hat folgende Bedeutung:

Generiere in der Form ein Ausgabeobjekt (*Label*), das die Inschrift (*Caption*) »Monatsgehalt« trägt (Feldbezeichnung, Feldname aus der Definitionsource an Feldnummer 11) und daneben ein fünf Zeichen großes Ausgabeobjekt (*Label*), welches den übergebenen Datenfeldinhalt rechtsbündig (definierter Feldtyp ist »l«) anzeigt.

Da der Form-Ausgabeinterpreter eine ganz entscheidende Eigenschaft besitzen soll, nämlich die, eine Ausgabeform mit einer variablen Anzahl von Ausgabefeldern zur Laufzeit generieren zu können, musste vom »herkömmlichen« Lösungsansatz Abstand genommen werden.

Würde man eine fixe Anzahl Objekte (*Label*) zur Datenausgabe in die Form einbetten, so wären, wenn weniger Ausgabefelder definiert als Objekte in der Form vorhanden, zu viele, sprich leere, Bezeichnungsfelder zu sehen. Auf der

Funktionsweise

anderen Seite kann es passieren, dass die fixe Anzahl Bezeichnungsfelder (*Label*) in der Form nicht ausreicht, da mehr zu generierende Ausgabefelder definiert als Ausgabeobjekte (*Label*) vorhanden sind.



Stellen Sie sich vor, es existieren in der Form Ihrer Adressverwaltung fix fünf Label, deren Inschriften (*Caption*) bei Laufzeit aus der Definitionsourcdatei mit den Feldbezeichnungen gefüllt werden.

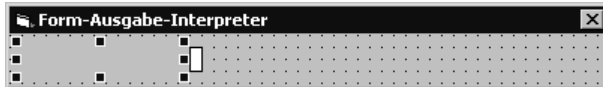
Werden nur vier Felder durch Vergabe der Feldnummer (Referenznummer) in der Interpretersourcdatei hinterlegt, würde beim Generieren der Ausgabeform ein Label leer sichtbar bleiben. Wären hingegen sechs Feldnummern in der Interpretersourcdatei hinterlegt, so könnte die sechste Feldbezeichnung gar nicht angezeigt werden, weil kein sechstes Label in der Form existent ist.

Eine weitere Möglichkeit wäre, da maximal 15 Ausgabefelder zu realisieren sind, genau 15 Label im Entwicklungsmodus in die Form einzubetten und die nicht benötigten zur Laufzeit auf nicht sichtbar zu setzen (*Visible = False*). Aber da wäre dann u.a. das Problem zu lösen, die im Entwicklungsmodus definierte Größe der Form, bedingt durch die höchste Anzahl an Bezeichnungsfelder, zur Laufzeit zu verringern usw.

dynamische Anlage von Objekten

Das Fazit aus dieser Problemanalyse lautet doch wohl, es muss auf irgendeine Art und Weise möglich sein, Bezeichnungsfelder (*Label*), allgemein Objekte, dynamisch erst bei Laufzeit der Applikation anzulegen.

Abbildung 6.75:
Formular des
Form-Ausgabe-
interpreters



Im Formular des Form-Ausgabeinterpreters (Abbildung 6.75) sind deshalb ausschließlich ein Steuerelement *Label* zur Ausgabe der Feldbezeichnung und ein weiteres Steuerelement *Label* zur Ausgabe des Datenfeldinhaltes, wir nennen sie der Einfachheit halber »Masterobjekte«, eingebettet.

Alle weiteren Bezeichnungsfelder (*Label*) werden bei Bedarf, der abhängig von der Anzahl der Ausgabefelder aus der Interpretersourcdatei ist, dynamisch durch Bildung von Instanzen angelegt. Was Instanzen sind, wie diese definiert und gebildet werden, wird im Kapitel Übung »Individuelle Programmschnellstartleiste (Application Caller)« hinreichend erklärt.

FAI.FRM FORM_LOAD

```
Übergabe$(1) = "Schlenker"
Übergabe$(2) = "Florian"
Übergabe$(3) = "Schülerweg 99"
Übergabe$(4) = "81250"
Übergabe$(5) = "München"
Übergabe$(6) = "08010 13234"
Übergabe$(7) = "Staubsauger AG"
Übergabe$(8) = "Augsburg"
```

```

Übergabe$(9) = "Verkauf"
Übergabe$(10) = "Vertriebs-Ing."
Übergabe$(11) = "5450"
Übergabe$(12) = "2450"
Übergabe$(13) = "3000"
Übergabe$(14) = "27"
Übergabe$(15) = "Skifahren, Tennis"
' Die Prozeduraufrufe
Call Dateien_Lesen
Call Original_Setzen
Call Objekte_Setzen
Call Form_Zentrieren

```

In der Ereignisprozedur *Form_Load* der Ausgabeform werden die individuell zu übergebenden Datenfeldinhalte in die Übergabetabelle *Übergabe\$()*, welche im BAS-Modul dimensioniert wurde, gestellt.

Danach erfolgen die standardisierten Prozeduraufrufe in dieser Reihenfolge und ihre Ausgabeform steht fix und fertig auf dem Bildschirm. Die Funktionalität dieser Prozeduren wurde im BAS-Modul *FAI.BAS* realisiert und wird nach dem Kurzüberblick der Funktionen erläutert.

Die Funktion *Dateien_Lesen* liest die Dateien Definitionsquelle *F11.TXT* und Interpretersource *F10.TXT* ein. *Original_Setzen* interpretiert die Interpreter- sowie Definitionsquelle und versorgt die Masterobjekte *Label*.

Kurzüberblick der Funktionen

Objekte_Setzen interpretiert die Sourcen und generiert dynamisch, durch Bildung von Instanzen, die angeforderten Ausgabeobjekte. *Form_Zentrieren* gibt die mit Ausgabeobjekten (*Label*) erzeugte Form bildschirmmittig aus.

```

Public Kürzel$(15)
Public Felder$(15)
Public K As Integer
Public Übergabe$(15)

```

**FAI.BAS
Deklarationen**

Generell wurden im BAS-Modul *FAI.BAS* die Tabellen *Kürzel\$()* – enthält die eingelesenen Feldnummern (Referenznummern) aus der Interpretersource –, *Felder\$()* – enthält die eingelesenen Feldbezeichnungen etc. aus der Definitionsquelle – und *Übergabe\$()*, enthält die auszugebenden Datenfeldinhalte, als *Public* (öffentlich) definiert. Die Variable *K* wird als Tabellenindex verwendet.

Um die Dateien Interpreter- und Definitionsquelle unter dem Laufwerk und Pfad öffnen zu können, in dem Sie den Form-Ausgabeinterpreter installiert haben, wird der Applikationspfad ausgelesen.

**FAI.BAS
Dateien_Lesen**

```

Sub Dateien_Lesen()
    aktppfad = App.Path
    If Right$(aktppfad, 1) <> "\" Then
        aktppfad = aktppfad + "\"
    End If
    ' Kürzeldatei

```

```

dbdat$ = aktpfad & "F10.TXT"
Dim dateinr As Integer
dateinr = FreeFile
' Kürzeldatei EINLESEN
Open dbdat$ For Input As dateinr
K = -1
While Not EOF(dateinr)
    K = K + 1
    ' Im Fehlerfall beenden
    If K > 15 Then End
    Line Input #dateinr, Kürzel$(K)
Wend
Close dateinr
' Felderdatei
dbdat$ = aktpfad & "F11.TXT"
dateinr = FreeFile
' Felderdatei EINLESEN
Open dbdat$ For Input As dateinr
F = 0
While Not EOF(dateinr)
    F = F + 1
    ' Im Fehlerfall beenden
    If F > 15 Then End
    Line Input #dateinr, Felder$(F)
Wend
Close dateinr
End Sub

```

Danach können die Einträge der beiden Dateien in die entsprechenden Tabellen zur weiteren Verarbeitung gefüllt werden. Ist der Lese-Index K größer als 15, findet der Form-Ausgabeinterpreter sein Ende.

FAI.BAS Original_Setzen

```

Sub Original_Setzen()
' 1. Originalobjekt versorgen
R = InStr(Kürzel$(0), "&")
Anker = Val(Mid$(Kürzel$(0), R + 1, 2))
fai.Label1(0).Caption = Trim$(Mid$(Felder$(Anker), 4, 17))
fai.Label1(0).Left = 120 + (R * 120)
' Bei Felddtyp "I" Ganzzahl rechtsbündige Anzeige
If Trim$(Mid$(Felder$(Anker), 25, 1)) = "I" Then
    fai.Label2(0).Alignment = 1
Else
    fai.Label2(0).Alignment = 0
End If
fai.Label2(0).Left = 1800 + (R * 120)
' Feldlänge
fai.Label2(0).Width = 180 +
(Val(Trim$(Mid$(Felder$(Anker), 23, 2))) * 90)
fai.Label2(0).Caption = Übergabe$(Anker)
End Sub

```


Die Feldbezeichnung wird entsprechend der gewünschten Feldnummer (Referenznummer der Interpretersource) in die Eigenschaft *Caption* des ersten Bezeichnungsfeldes (Masterobjekt), das den Namen *Label1* indiziert mit 0 trägt, gefüllt.

Danach wird das *Label1*, abhängig von der Position der Feldnummer in der Interpretersource, über die Eigenschaft *Left* ausgerichtet.

Der übergebene Datenfeldinhalt wird ebenfalls entsprechend der gewünschten Feldnummer (Referenznummer der Interpretersource) in die Eigenschaft *Caption* des zweiten Bezeichnungsfeldes (Masterobjekt), das den Namen *Label2* indiziert mit 0 trägt, gefüllt.

Über die Eigenschaft *Alignment* wird der auszugebende Datenfeldinhalt entsprechend dem definierten Feldtyp entweder rechts- oder linksbündig im *Label2* angezeigt.

Danach wird die Größe des *Label2*, entsprechend der definierten Feldlänge über die Eigenschaft *Width* und die Bildschirmposition über die Eigenschaft *Left* festgelegt.

```
Sub Objekte_Setzen()
  For i = 1 To K
    R = InStr(Kürzel$(i), "&")
    Anker = Val(Mid$(Kürzel$(i), R + 1, 2))
    Load fai.Label1(i) ' Neues Objekt
    Load fai.Label2(i) ' Neues Objekt
    fai.Label1(i).Top = fai.Label1(i - 1).Top + 360
    fai.Label2(i).Top = fai.Label2(i - 1).Top + 360
    fai.Label1(i).Visible = True ' Neuen Button anzeigen
    fai.Label2(i).Visible = True ' Neuen Button anzeigen
    fai.Label1(i).Caption = Trim$(Mid$(Felder$(Anker), 4, 17))
    fai.Label1(i).Left = 120 + (R * 120)
    ' Bei Feldtyp "I" Ganzzahl rechtsbündige Anzeige
    If Trim$(Mid$(Felder$(Anker), 25, 1)) = "I" Then
      fai.Label2(i).Alignment = 1
    Else
      fai.Label2(i).Alignment = 0
    End If
    fai.Label2(i).Left = 1800 + (R * 120)
    fai.Label2(i).Width = 180 + (Val(Trim$(Mid$(Felder$(Anker), 23,
2))) * 90)
    fai.Label2(i).Caption = Übergabe$(Anker)
    ' Formgröße anpassen
    fai.Height = fai.Height + 360
  Next i
End Sub
```

FAI.BAS
Objekte_Setzen

Werden weitere Ausgabeobjekte benötigt, so werden diese in einer Schleife, die die Anzahl Feldnummern aus der Interpretersource repräsentiert (*K*), durch Instanzenbildung erzeugt.

Neue Bezeichnungsfelder (*Label*) werden mit der Methode *Load* unter den darüber liegenden Bezeichnungsfeldern (*Label*) durch Indizierung dynamisch angelegt, angezeigt (Eigenschaft *Visible = True*) und mit den Feldtexten bzw. Feldinhalten versorgt (Eigenschaft *Caption =.....*).

Danach muss noch die Größe der Form angepasst werden, damit die neuen Bezeichnungsfelder überhaupt in der Form sichtbar werden.

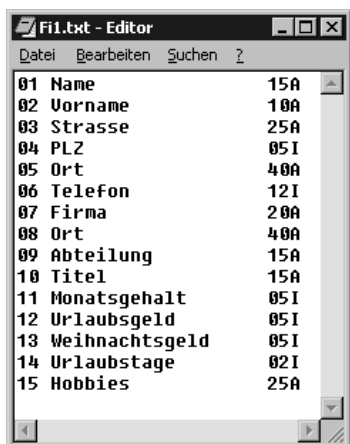
FAI.BAS Form_Zentrieren

```
' Form zentrieren
Fai.Left = (Screen.Width - Fai.Width) / 2
Fai.Top = (Screen.Height - Fai.Height) / 2
```

Damit die generierte Ausgabeform nicht irgendwo auf dem Bildschirm erscheint, wird diese zentriert gesendet.

Start Der Start des Programms *FAI.EXE* dokumentiert umfassend die Leistungsfähigkeit des Form-Ausgabeinterpreters. Das Formularmodul *FAI.FRM* soll Ihnen als Mustermodul, zum Ansteuern des Basicmoduls *FAI.BAS* aus Ihren eigenen Visual Basic-Applikationen, dienlich sein.

Abbildung 6.76:
Definitionsourcedatei *F11.TXT*



Index	Name	Width
01	Name	15A
02	Vorname	10A
03	Strasse	25A
04	PLZ	05I
05	Ort	40A
06	Telefon	12I
07	Firma	20A
08	Ort	40A
09	Abteilung	15A
10	Titel	15A
11	Monatsgehalt	05I
12	Urlaubsgeld	05I
13	Weihnachtsgeld	05I
14	Urlaubstage	02I
15	Hobbies	25A

Bevor Sie nun den Form-Ausgabeinterpreter starten, können Sie entsprechend Ihren Anforderungen und Wünschen Ihre individuellen Feldbeschreibungen in der Definitionsourcedatei *F11.TXT* mit dem Windows-Editor *Notepad* hinterlegen (Abbildung 6.76).

Welche Ausgabefelder tatsächlich zur Anzeige kommen sollen, können Sie nun individuell in der Interpretersourcedatei *F10.TXT* mit dem Windows-Editor *Notepad* hinterlegen (Abbildung 6.77).

Werden beide Dateien gespeichert und der Form-Ausgabeinterpreter danach gestartet, so generiert er daraus die folgende Ausgabemaske (Abbildung 6.78).

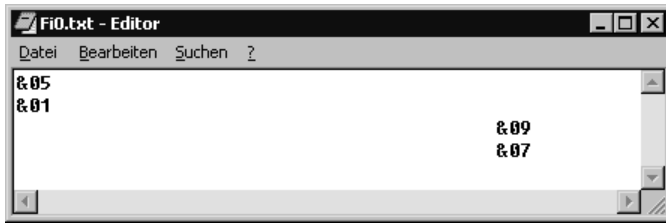


Abbildung 6.77:
Interpretersource-
datei FIO.TXT



Abbildung 6.78:
Entsprechend
generiertes
Ausgabeformular

Passen Sie Ihre individuelle Interpretersourcedatei *FIO.TXT* durch Ändern, Löschen oder Hinzufügen von Feldnummern (Referenznummern) an, so generiert das Tool bei Start daraus erneut das entsprechende Ausgabeformular.

Ändern Sie Feldbeschreibungen in der Definitionssourcedatei *F11.TXT*, so wird diese Modifikation, ebenfalls ohne Programmieraufwand, nach Aufruf des Form-Ausgabeinterpreters aktiv (Interpretation bei Laufzeit).

Außerdem ist es, wie schon kurz erwähnt, mit dieser externen Definitionssource möglich, multilinguale Anwendungen zu erzeugen, indem ausschließlich die Sourcedatei *F11.TXT* in die entsprechende Landessprache übersetzt wird.

6.11 Das Steuerelement »TextBox«

Die TextBox kann einen Text anzeigen, der während des Programmlaufs vom Benutzer geändert oder eingegeben werden kann. Auch sein Aussehen ist durch Eigenschaften beeinflussbar.

6.11.1 Übung: Eingabe von Passwörtern

Über eine Befehlsschaltfläche (*CommandButton*) soll das Passwort, sprich die Eingabe, aus dem Textfeld in einem Bezeichnungsfeld (*Label*) ausgegeben werden (Abbildung 6.80). Dabei soll, wie in der Abbildung 6.79 zu sehen, der in der Textbox eingegebene Text zeichenweise als »#« visualisiert werden, damit das Passwort quasi nicht ausgespäht werden kann.



Abbildung 6.79:
Die getätigte
Eingabe wird mit
»#« visualisiert

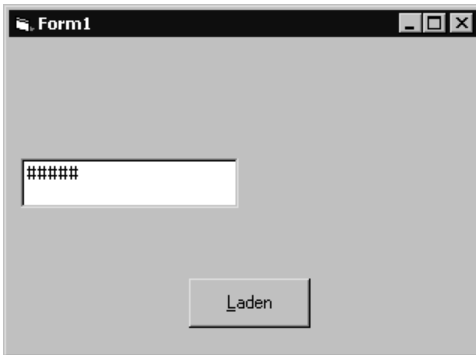
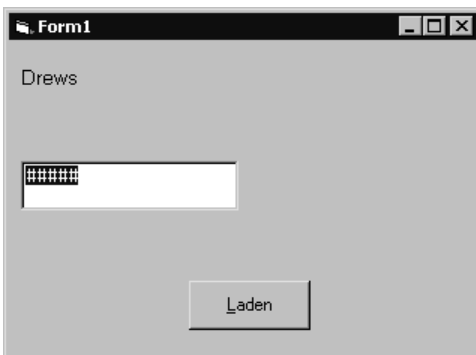


Abbildung 6.80:
Das Bezeichnungs-
feld ist mit dem
Text (Passwort)
aus dem Textfeld
geladen



Lösung

Dass Textfelder allgemein zur Dateneingabe verwendet werden, ist wohl hinreichend bekannt, deshalb zielt diese Übung auf die Verwendung der *TextBox* als Eingabemöglichkeit für Passwörter ab.

In der Eigenschaft *PasswordChar* kann man ein Zeichen angeben, das anstelle der tatsächlich eingegebenen Zeichen in der *TextBox* angezeigt werden soll (Abbildung 6.81).

Der eingegebene Text steht für Sie als Programmierer weiterhin im Klartext in der Eigenschaft *Text* zur Verfügung. Somit können Passwörter oder Geheimzahlen verdeckt abgefragt werden.

Damit bei Mausklick auf den Button »Laden« auch der Text, der im Textfeld eingegeben wurde, im Label erscheint, ist die Ereignisprozedur *Click* des *CommandButtons* wie folgt anzupassen.

```
Private Sub Command1_Click()  
    Label1.Caption = Text1.Text  
End Sub
```



Abbildung 6.81:
Eigenschaft `PasswordChar` auf »#«
gesetzt

Die in der Textbox getätigte Eingabe wird automatisch in der Eigenschaft `Text` gehalten und somit ist für die Ausgabe im Bezeichnungsfeld ausschließlich der Inhalt der Eigenschaft `Text` der Eigenschaft `Caption` (Inschrift, Überschrift) des Bezeichnungsfeldes `Label1` zuzuweisen.

```
Private Sub Command1_Click()
    Label1.Caption = Text1.Text
    Text1.SelStart = 0
    Text1.SelLength = Len(Text1.Text)
    Text1.SetFocus
End Sub
```

In der Ereignisprozedur `Click` ist noch ein weiteres »Schmankerl« enthalten. Nach dem Laden des Passwortes in das Bezeichnungsfeld soll, wie in der Abbildung 6.80 zu sehen, das Textfeld aktiviert und die Eingabe im Textfeld markiert sein, damit sofort eine neue Eingabe erfolgen kann.

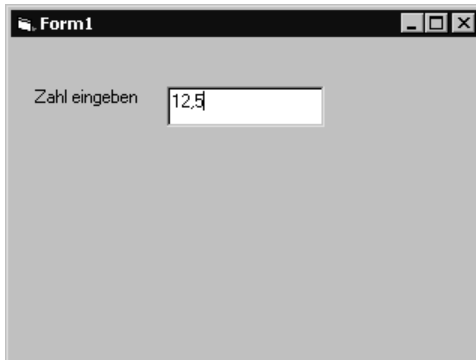
Damit die gesamte Eingabe im Textfeld markiert ist, setzen Sie den Beginn durch die Eigenschaft `SelStart= 0` und das Ende durch die Eigenschaft `SelLength = Gesamtlänge der Eingabe` (Über die Funktion `Len`). Dadurch ist im Textfeld alles markiert. Zum Schluss wird durch die Methode `SetFocus` der Fokus auf das Steuerelement Textfeld `Text1` bewegt.



6.11.2 Übung: Numerische Eingabeprüfung

Das Steuerelement *Textfeld* ist von seiner Art her ein Objekt, das grundsätzlich bei der Eingabe jedes Zeichen (Ziffer, Buchstabe und Sonderzeichen) aufnimmt. Sehr häufig werden aber in Applikationen in manchen Textfeldern nur numerische Eingaben erwartet, also ausschließlich Ziffern.

Abbildung 6.82:
Nur numerische
Eingabe möglich



Damit der Anwender entsprechend gelenkt wird und Fehleingaben, z. B. alpha-nummerische oder alphabetische Eingaben, möglichst ausgeschlossen werden, ist eine allgemeingültige, globale Prozedur mit den folgenden Leistungsmerkmalen zu entwickeln:

- ▶ Die Prozedur soll in einem Basic-Modul, das jederzeit in andere Visual Basic-Applikationen eingebunden werden kann, enthalten sein.
- ▶ Die Prozedur soll für jede Textbox verwendet werden können.
- ▶ Die Prüfung auf numerisch hat sofort bei der Eingabe, sprich bei Tastendruck, im Textfeld zu erfolgen (Abbildung 6.82).
- ▶ Zusätzlich soll ein eingegebener Dezimalpunkt automatisch in ein Dezimalkomma umgewandelt und im Textfeld visualisiert werden (Abbildung 6.82).

Lösung

Tätigen Sie in ein Textfeld eine Eingabe, wird immer für jedes gedrückte Zeichen die Ereignisprozedur *KeyPress* aufgerufen. Als Parameter bekommt die Prozedur den ASCII-Wert (die ASCII-Tabelle enthält den PC-Zeichensatz) des gedrückten Zeichens (jedes Zeichen hat einen internen numerischen Wert zwischen 0 und 255, z. B. hat der Buchstabe A den ASCII-Wert 65) übermittelt.

```
Private Sub Text1_KeyPress(KeyAscii As Integer)
    Call Tasten_Pruef(KeyAscii)
End Sub
```

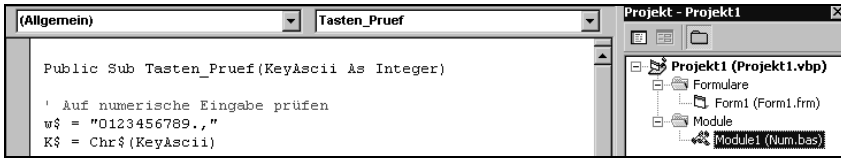


Abbildung 6.83:
Prüfprozedur in
einem Basic-Modul
verankern

Also lässt sich genau in dieser Prozedur prüfen, ob das eingegebene Zeichen numerisch ist. In diese Ereignisprozedur *KeyPress* legen wir den Aufruf unserer allgemeingültigen, globalen Prüfprozedur, die in einem Basic-Modul zu erstellen ist (Abbildung 6.83), und geben dieser natürlich auch den ASCII-Wert des im Textfeld gedrückten Zeichens mit.

```
Public Sub Tasten_Pruef(KeyAscii As Integer)
' Auf numerische Eingabe prüfen
w$ = "0123456789.,"
K$ = Chr$(KeyAscii)
If InStr(w$, K$) = 0 Then
    Beep
    KeyAscii = 0
Else
    If K$ = "." Then
        KeyAscii = Asc(",")
    End If
End If
End Sub
```

Alle gültigen Zeichen die im Textfeld eingegeben werden dürfen, stehen einzeln hintereinander im String *w\$*. Das Zeichen, das im Textfeld eingegen wurde (z.B. »A«), liegt uns ja nur als numerischer ASCII-Wert in der Übergabe-Variablen *KeyAscii* vor (z.B. 65) und wird deshalb mit der Funktion *CHR\$* in ein »normales« Zeichen in die Variable *K\$* (enthält dann das tatsächlich eingegebene Zeichen, z.B. »A«) umgewandelt.

Mit der Funktion *InStr* kann dann geprüft werden, ob das im Textfeld gedrückte Zeichen *K\$* numerisch gültig ist, also somit im Gültigkeitsstring *w\$* enthalten ist. Ist dies nicht der Fall, wird die Prozedur mit einem Tonsignal (*Beep*) verlassen, andernfalls wird geprüft, ob ein eventuell eingegebener Dezimalpunkt in ein Dezimalkomma umgewandelt werden soll.

6.11.3 Übung: Ein kleiner Editor gefällig

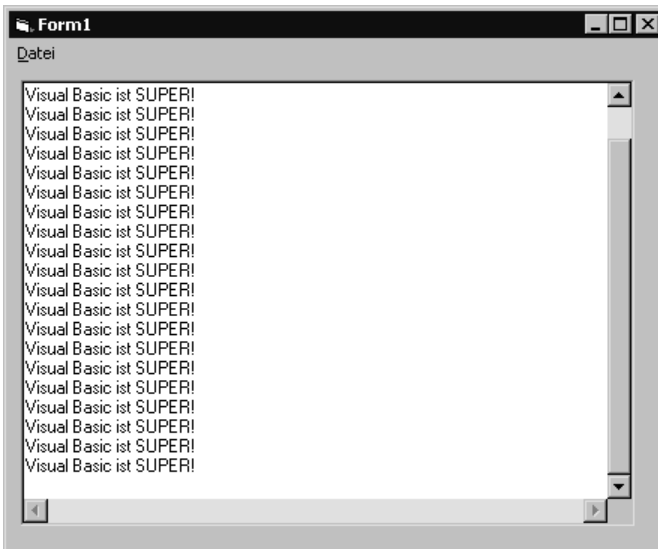
Auf Anforderung soll es, wie in Abbildung 6.84 zu sehen, möglich werden, im Startformular einen »kleinen« funktionstüchtigen Editor (Abbildung 6.85) bereitzustellen.



Abbildung 6.84:
Menü zum Aktivieren
oder Deaktivieren
des Editors



Abbildung 6.85:
Aktivierter Editor



Lösung

Die *TextBox* gehört zur Gruppe äußerst wichtiger Steuerelemente. Die grundlegende Funktion einer *TextBox* wird durch die Eigenschaft *MultiLine* bestimmt. Diese Eigenschaft ermöglicht Texteingaben auch über mehrere Zeilen hinweg.

Ist der Wert *False*, so kann nur eine Zeile eingegeben und dargestellt werden. Mit der *MultiLine*-Eigenschaft auf *True* werden zudem einige weitere Eigenschaften aktiv, die zuvor ohne Auswirkung waren:

Dies sind die *Alignment*- und die *Scrollbars*-Eigenschaft. *Alignment* kann den Text links, rechts oder zentriert formatieren.

Scrollbars werden immer angezeigt, jedoch erst dann aktiviert, wenn der Text breiter ist oder mehr Zeilen hat, als das Steuerelement in seiner aktuellen Größe aufnehmen kann.

```
Private Sub Form_Load()  
    Text1.Visible = False  
End Sub
```

Damit beim ersten Anzeigen des Formulars der Editor nicht sichtbar ist, wird in der Ereignisprozedur *Form_Load* die Eigenschaft *Visible* des Textfeldes auf *False* gesetzt.

```
Private Sub aktivieren_Click()  
    Text1.Visible = True  
End Sub
```

Das Aktivieren des Editors über die Menüauswahl hat das Setzen der Eigenschaft *Visible* des Textfeldes auf *True* in der Ereignisprozedur *Aktivieren_Click* zur Folge.

```
Private Sub deaktivieren_Click()  
    Text1.Text = Empty  
    Text1.Visible = False  
End Sub
```

In der Ereignisprozedur *Deaktivieren_Click* wird die Texteingabe geleert (*Empty*) und das Textfeld, unser Editor, als nicht sichtbar dargestellt (*Visible=False*).

6.11.4 Übung: Entwicklung eines Form-Eingabeinterpreters

Da der Form-Eingabeinterpreter von der Art, der Logik und dem internen Aufbau sehr ähnlich wie der Form-Ausgabeinterpreter funktioniert, finden Sie im Kapitel zum »Form-Ausgabeinterpreter« ausführliche Beschreibungen zum Lösungsansatz gerade auch in Bezug auf die Themen »Dynamische Anlage von Objekten und Instanzenbildung«.

Die »umfangreiche« Eingabemaske der in Visual Basic programmierten Adress-Verwaltung ist für einen Neukunden anzupassen.

Adress-Eingabe

Anrede : *05A

Name : *20A

Vorname : *20A

Straße : *20A



Standardbild
(komprimiert)

Landeskz : *01A

Ort : *25A

PLZ : *04I

Telefon : *10A

Postfach : *15A

Legende

- ▶ *NNA oder *NNI.
- ▶ »NN« steht für eine Zahl, die die Eingabefeldlänge repräsentiert.
- ▶ »A« bedeutet Feldtyp alphanumerisch,
- ▶ »I« bedeutet Feldtyp numerisch (Ganzzahl).

fiktive Anforderung

- ▶ Der Standard-Datensatz-Aufbau kann ohne Einschränkung verwendet werden (Datenbasis bleibt bestehen).
- ▶ Die Eingabemaske ist wie folgt zu ändern :

Adress-Eingabe

Anrede : *06A

Vorname : *25A

Name : *25A

LKZ : *03A

PLZ : *05I

Ort : *30A

Straße: *20A

Telefon : *15I

Da die zu sendende Form mit seinen Ein-/Ausgabeobjekten und die Eingabesteuerung innerhalb des Programms »hart« kodiert wurden (im Entwicklungsmodus wurde die Anzahl Ein-/Ausgabeobjekte auch mit den Beschriftungen der Label etc. statisch festgelegt), bleibt keine andere Wahl, als das Formular entsprechend zu modifizieren und einen neuen kundenindividuellen Programmstand zu erzeugen.

Fazit

Der zwangsläufig entstehende Programmieraufwand (kostenintensiv) summiert sich durch die Anforderungshäufigkeit.

Aus diesem Grund soll ein Visual Basic-Modul entwickelt werden, mit dem wir dann in der Lage sind, die Bildschirmeingabefelder einer Form, die mit einem beliebigen Texteditor definiert werden können, erst bei Laufzeit der Visual Basic-Applikation zu interpretieren und anzuzeigen.

Dieses Modul soll außerdem so variabel und dynamisch sein, dass es in jede Visual Basic-Anwendung eingebunden und über die Anweisung *CALL* mit Parameterübergabe aufgerufen werden kann.

- ▶ Einfach zu handhabendes Interpreter- und Lösungskonzept.
- ▶ Über einen beliebigen Windows-Editor, z.B. Notepad, sollen die einzugebenden Felder (Feldinhalte und Feldbezeichnungen) einer Eingabemaske (Form) als Text (Definitionsquelle), ohne Visual Basic-Anweisungen, hinterlegt werden können.
- ▶ Zusätzlich sollen in der Definitionsquelle die Feldlänge und der Typ der einzugebenden Datenfelder deklariert werden können, damit bei Sendeanforderung der Form die Größe der zu generierenden Eingabeobjekte (Textboxen) dynamisch bestimmt und Dateninhalte von numerisch definierten Eingabeobjekten automatisch geprüft werden können.
- ▶ Welche in der Definitionsquelle hinterlegten Eingabefelder nun tatsächlich in der zu sendenden Form angezeigt werden, ist in einer weiteren Textdatei, der Interpretersource, die ebenfalls ganz einfach über einen Windows-Editor, z.B. Notepad, erstellt werden kann, zu bestimmen.
- ▶ Zusätzlich soll in der Interpretersource bestimmt werden, an welcher Position innerhalb einer Zeile das zu generierende Ausgabeobjekt (Label) für die Feldbezeichnung und deren Eingabeobjekt (Textbox) zur Dateneingabe stehen soll.
- ▶ Dieses Modul soll in dieser Version ausschließlich bis zu 15 Eingabefelder interpretieren, soll aber durch entsprechende Dimensionierung im Modul verändert werden können.
- ▶ Zeitgewinn bei der Programmentwicklung und -anpassung.
- ▶ Die Eingabemaske ist vom Programm physisch getrennt.
- ▶ Bei Bildänderungen wird es daher nicht nötig, die Applikation zu modifizieren, sondern ausschließlich die entsprechende Definition- Interpretersource.
- ▶ Entwicklungs-, Compile- und Linkläufe entfallen.
- ▶ Kundenindividuelle Wünsche über Feldplatzierungen innerhalb einer Form führen nicht zwangsläufig zu kundenindividuellen Programmständen.
- ▶ Minimierung des Programmieraufwands.
- ▶ Aufbauinterpretation erst bei Sendeanforderung der Form.
- ▶ Integration in Anwendungen jeder Art durch das Einbinden als Modul.
- ▶ Einfache Ansteuerung.
- ▶ Da die Feldbezeichnungen wie z.B. Name, Ort etc. nicht in der Form eines Visual Basic-Programms »hart« hinterlegt sind, sondern vom Programm ausgelagert in einer ASCII/ANSI-Datei stehen, lässt sich die Visual Basic Applikation ebenfalls sehr leicht internationalisieren.

Leistungsmerkmale, Leistungsbeschreibungen und Restriktionen

- ▶ Die Visual Basic-Anwendung kann somit ohne großen Aufwand mehrsprachig ausgeliefert werden, ohne dabei das Programm modifizieren zu müssen.
- ▶ Nicht die Applikation, sondern ausschließlich die Definitionsource muss mit Hilfe eines Übersetzungstools in die gewünschte Landessprache übersetzt werden. Das Visual Basic-Programm muss nicht angepasst werden.

Ablauflogo

1. Definitionsource, in der alle Eingabefelder global definiert werden, mit einem ASCII/ANSI-Texteditor, z.B. Notepad, erstellen.
2. Interpretersource, in der die tatsächlich zu sendenden Eingabefelder definiert werden (Eine Teilmenge aus der Menge, die in der Definitionsource hinterlegt wurde), mit einem ASCII/ANSI-Texteditor erstellen.
3. Formmodul in Ihre Visual Basic-Anwendung einbinden.
4. Basicmodul in Ihre Visual Basic-Anwendung einbinden.
5. Parameter übergeben und Form-Eingabeinterpreter aufrufen.
6. Definition- und Interpretersourcedateien werden geladen, interpretiert und die Form entsprechend aufgebaut.

Lösung

Damit der Form-Eingabeinterpreter seine Arbeit aufnehmen kann, erstellen oder modifizieren wir mit einem beliebigen Editor (z.B. Notepad), der im ASCII/ANSI-Format speichert, folgende Dateien:

Definitionsource FI1.TXT

In diesem File definieren Sie Ihre individuellen Eingabefelder. Der Form-Eingabeinterpreter liest die Definitionsource und erhält Informationen darüber, wie die zu generierenden Feldbezeichnungen (Feldnamen) lauten und in welcher Größe die Eingabeobjekte (*Textboxen*) für die Dateneingabe aufzubauen sind.

In dieser Datei hinterlegen Sie alle Eingabefelder auf Basis der Daten Ihrer Visual Basic-Anwendung.

Stellen Sie sich dazu vor, Ihr Visual Basic-Programm ist eine Adressverwaltung mit einer Datenbasis von zehn Dateifeldern. So sollten auch alle zehn Dateifelder als Eingabefelder in die Definitionsource eingestellt werden. Welche Eingabefelder später in der Form tatsächlich angezeigt werden, ist in der Interpretersource zu definieren. Damit ist eine vollkommene Variabilität gewährleistet.

Konvention mit Beispiel ab Zeile 1 Spalte 1 beginnend :

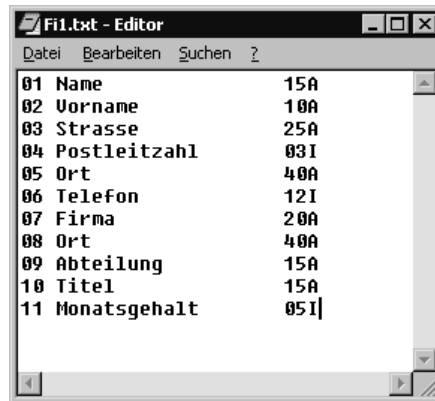
Konvention

Spalte 001 – 002: Feldnummer, die zur Referenzierung benötigt wird.

Spalte 004 – 020: Feldbezeichnung (Feldname), die als Caption in ein Ausgabeobjekt (Label) einzustellen ist.

Spalte 023 – 024: Eingabefeldlänge, um das Eingabeobjekt (Textbox) zur Dateneingabe entsprechend groß, dynamisch zu generieren.

Spalte 025: Feldtyp. Entweder »A« = alphanummerischer Felddatentyp oder »I« = numerischer Felddatentyp (Ganzzahl). Ist das Eingabefeld als numerisch deklariert, werden nur Ziffern im Bereich von 0 bis 9 und Blank zur Eingabe zugelassen.



Feldnummer	Name	Feldtyp
01	Name	15A
02	Vorname	10A
03	Strasse	25A
04	Postleitzahl	03I
05	Ort	40A
06	Telefon	12I
07	Firma	20A
08	Ort	40A
09	Abteilung	15A
10	Titel	15A
11	Monatsgehalt	05I



Abbildung 6.86:
Definitionsquelle
F11.TXT

Die Definitionsquelle *F11.TXT* darf max. 15 Zeilen (Einträge) enthalten, das heißt der Form-Eingabeinterpreter kann höchstens 15 Eingabefelder in einer Form generieren.

In dieser Datei hinterlegen Sie, durch Angabe der Referenznummer, ausschließlich die Felder aus der gesamten Menge aller zur Verfügung stehenden Eingabefelder (definiert in der Definitionsquelle), die nun tatsächlich in der Form als Eingabeobjekte (*Textboxen*) generiert werden sollen.

Der Form-Eingabeinterpreter liest die Interpretersource und erhält durch die Referenznummer Informationen darüber, welche Eingabefelder aus dem gesamten Felderpool (Definitionsquelle) an welcher Position innerhalb der Form angezeigt werden sollen.

In dieser Datei können Sie somit eine Teilmenge aus der definierten Menge Eingabefelder hinterlegen. Stellen Sie sich dazu vor, Ihr Visual Basic-Programm ist eine Adressverwaltung mit einer Datenbasis von zehn Dateifeldern, die auch alle zehn als Eingabefelder in die Definitionsquelle eingestellt wurden.

Wie viel und welche Eingabefelder aus diesen zehn nun wirklich in der Eingabeform angezeigt werden, definieren Sie in der Interpretersource. Damit ist auch hier vollkommene Variabilität gewährleistet.

Konvention mit Beispiel ab Zeile 1 beginnend und weitgehendst spaltenunabhängig:

Angabe der Feldnummer mit vorangestelltem »&«, die zur Referenzierung auf die in der Definitionsquelle hinterlegten Felder benötigt wird.

Interpretersource F10.TXT

Konvention

Praxisbeispiel 1 &01
 &02
 &03
 &04
 &05

Die Interpretersource *FIO.TXT* darf max. 15 Zeilen (Einträge) enthalten, das heißt der Form-Eingabeinterpreter kann höchstens 15 Eingabefelder in einer Form generieren.

Praxisbeispiel 2 &05
 &01
 &10
 &11
 &09
 &07

Bedeutung

Abbildung 6.87:
 Interpretiertes und
 generiertes Praxis-
 beispiel 1

The screenshot shows a window titled "Form-Eingabe-Interpreter" with a grey background. It contains five input fields arranged vertically, each with a label to its left. The labels and their corresponding values are: "Name" with "Schlenker", "Vorname" with "Floria", "Strasse" with "Schülerweg 99", "Postleitzahl" with "812", and "Ort" with "München".

Durch das Praxisbeispiel 1 werden die ersten fünf Eingabefelder, die in der Definitionsquelle zur Verfügung stehen, in Reihenfolge als Eingabeobjekte untereinander in der Form generiert (Abbildung 6.87), während im Praxisbeispiel 2 deutlich wird, dass auch eine Teilmenge der Eingabefelder in irgendeiner Reihenfolge als Eingabeobjekte versetzt in der Form generiert werden können (Abbildung 6.88).

Abbildung 6.88:
 Interpretiertes und
 generiertes Praxis-
 beispiel 2

The screenshot shows a window titled "Form-Eingabe-Interpreter" with a grey background. It contains six input fields arranged vertically, each with a label to its left. The labels and their corresponding values are: "Ort" with "München", "Name" with "Schlenker", "Titel" with "Vertriebs-Ing.", "Monatsgehalt" with "5450", "Abteilung" with "Verkauf", and "Firma" with "Staubsauger AG".

Was bedeutet aber nun zum Beispiel der Eintrag »&11« in der Interpretersource des Praxisbeispiels 2?

Dieser Eintrag, der als Feldnummer interpretiert wird, referenziert mit seinem Wert auf den sich hinter der Feldnummer verbergenden Inhalt der Definitionsource und besagt:

Generiere in der Form ein Ausgabeobjekt (*Label*), das die Inschrift (*Caption*) »Monatsgehalt« trägt (Feldbezeichnung, Feldname aus der Definitionsource an Feldnummer 11) und daneben ein fünf Zeichen großes Eingabeobjekt (*Textbox*), das ausschließlich fünf Zeichen (Blanks oder Ziffern zwischen 0 und 9) entgegennimmt (Definierter Eingabefeldtyp ist »I«).

Funktionsweise

In der Ereignisprozedur *Form_Load* der Eingabeform werden die eventuell zu übergebenden Eingabefeldinhalte (individuell) in die Übergabetabelle *Übergabe\$()*, welche im BAS-Modul dimensioniert wurde, gestellt.

FEI.FRM
FORM_LOAD

Danach erfolgen diese standardisierten Prozeduraufrufe

- ▶ Call *Dateien_Lesen*
- ▶ Call *Original_Setzen*
- ▶ Call *Objekte_Setzen*
- ▶ Call *Form_Zentrieren*

in dieser Reihenfolge und Ihre Eingabeform steht fix und fertig auf dem Bildschirm.

Die Funktion *Dateien_Lesen* liest die Dateien Definitionsource *F10.TXT* und Interpretersource *F10.TXT* ein. *Original_Setzen* interpretiert die Interpreter- sowie Definitionsource und versorgt die Masterobjekte *Label* und *Textbox*. *Objekte_Setzen* interpretiert die Sourcen und generiert dynamisch, durch Bildung von Instanzen, die angeforderten Ein-/Ausgabeobjekte. *Form_Zentrieren* gibt die mit Ein-/Ausgabeobjekten erzeugte Form bildschirmmittig aus.

Kurzüberblick der Funktionen

Generell wurden im BAS-Modul *FEI.BAS* public (öffentlich) definiert:

FEI.BAS
Deklarationen

- ▶ Die Tabelle *Kürzel\$()* enthält die eingelesenen Feldnummern (Referenznummern) aus der Interpretersource,
- ▶ die Tabelle *Felder\$()* enthält die eingelesenen Feldbezeichnungen etc. aus der Definitionsource und
- ▶ die Tabelle *Übergabe\$()*, enthält die auszugebenden Feldinhalte für die Eingabefelder.
- ▶ Die Variable *K* wird als Tabellenindex verwendet.

Um die Dateien Interpreter- und Definitionsource unter dem Laufwerk und Pfad öffnen zu können, in dem Sie den Form-Eingabeinterpreter installiert haben, wird der Applikationspfad *APP.PATH* ausgelesen.

FEI.BAS
Dateien_Lesen

Danach können die Einträge der beiden Dateien in die entsprechenden Tabellen zur weiteren Verarbeitung gefüllt werden.

FEI.BAS Original_Setzen

Die Feldbezeichnung wird entsprechend der gewünschten Feldnummer (Referenznummer der Interpretersource) in die Eigenschaft *Caption* des ersten Bezeichnungsfeldes (Masterobjekt), das den Namen *Label1* indiziert mit 0 trägt, gefüllt.

Danach wird das *Label1*, abhängig von der Position der Feldnummer in der Interpretersource, über die Eigenschaft *Left* ausgerichtet.

Der übergebene Datenfeldinhalt wird ebenfalls entsprechend der gewünschten Feldnummer (Referenznummer der Interpretersource) in die Eigenschaft *Text* der Textbox (Masterobjekt), die den Namen *Text1* indiziert mit 0 trägt, gefüllt.

Über die Eigenschaft *MaxLength* wird die Eingabelänge in der Textbox, entsprechend der in der Definitionsquelle hinterlegten Eingabefeldlänge, begrenzt.

Danach wird die Größe der Textbox, entsprechend der Eingabefeldlänge, über die Eigenschaft *Width* und die Bildschirmposition über die Eigenschaft *Left* festgelegt.

FEI.BAS Objekte_Setzen

Werden weitere Ein-/Ausgabeobjekte benötigt, so werden diese in einer Schleife, die die Anzahl Feldnummern aus der Interpretersource repräsentiert (Variable *K*), durch Instanzenbildung erzeugt.

Die Eingabemaske entsteht

Neue Steuerelemente *Label/Textboxen* werden mit der Methode *Load* unter den darüber liegenden Objekten durch Indizierung dynamisch angelegt, angezeigt (*Visible = True*) und mit den Feldtexten bzw. Feldinhalten versorgt (Eigenschaften *Caption =....., Text =*).

Danach muss noch die Größe der Form angepasst werden, damit die neuen Eingabefelder überhaupt in der Form sichtbar werden.

FEI.FRM KeyPress

```
If Text1(Index).Tag = "I" Then
  If KeyAscii = 32 Then Exit Sub
  ' Backspace
  If KeyAscii = 8 Then Exit Sub
  If KeyAscii < 48 Or KeyAscii > 57 Then
    Beep
    KeyAscii = 0
  End If
End If
```

Ebenfalls interessant an der objektindizierten Vorgehensweise ist, dass es nicht für »n« Textboxen »n« Eventprozeduren gibt, sondern ausschließlich eine, in der allerdings auf alle Textboxen einzeln über den Index Bezug genommen werden kann. So wird in der Ereignisprozedur *KeyPress* bei definiertem Eingabefeldtyp »I« die Eingabe auf numerisch geprüft.

Start

Der Start des Programms *FEI.EXE* dokumentiert umfassend die Leistungsfähigkeit des Form-Eingabeinterpreters. Das Formularmodul *FEI.FRM* soll Ihnen als Mustermodul, zum Ansteuern des Basicmoduls *FEI.BAS* aus Ihren eigenen Visual Basic-Applikationen, dienlich sein.

Bevor Sie den Form-Eingabeinterpreter starten, können Sie entsprechend Ihren Anforderungen und Wünschen Ihre individuellen Feldbeschreibungen in der Definitionsourcdatei *F11.TXT* mit einem Windows-Editor, z. B. *Notepad*, hinterlegen.

Welche Eingabefelder tatsächlich zur Anzeige kommen sollen, können Sie nun individuell in der Interpretersourcdatei *F10.TXT* mit dem Windows-Editor hinterlegen.

The screenshot shows a window titled "Form-Eingabe-Interpreter" with a close button (X) in the top right corner. The window contains a list of labels on the left and corresponding text input boxes on the right. The data entered in the boxes is as follows:

Name	Schlenker
Vorname	Florian
Strasse	Schülerweg 99
Postleitzahl	812
Ort	München
Telefon	08010 13234
Firma	Staubsauger AG
Ort	Augsburg
Abteilung	Verkauf
Titel	Vertriebs-Ing.
Monatsgehalt	5450
Urlaubsgeld	2450
Weihnachtsgeld	3000
Urlaubstage	27
Hobbies	Skifahren, Tennis

Abbildung 6.89:
Entsprechend
generiertes
Eingabeformular

Werden beide Dateien gespeichert und der Form-Eingabeinterpreter danach gestartet, so generiert er daraus die entsprechende Eingabemaske (Abbildung 6.89).

Passen Sie Ihre individuelle Interpretersourcdatei *F10.TXT* durch Ändern, Löschen oder Hinzufügen von Feldnummern (Referenznummern) an, so generiert der Form-Eingabeinterpreter bei Start daraus erneut die entsprechende Eingabeform.

Ändern Sie Feldbeschreibungen in der Definitionsourcdatei, so wird diese Modifikation ebenfalls ohne Programmieraufwand nach Aufruf des Form-Eingabeinterpreters aktiv (Interpretation bei Laufzeit).

Mit diesen beiden Interpretern, Form-Ausgabeinterpreter und Form-Eingabeinterpreter, stehen Ihnen zwei Modulkomponenten zur Verfügung, mit denen Sie sehr komfortabel komplexe Ausgabe- oder Eingabemasken ohne nennenswerten Programmieraufwand »vollkommen« variabel aus Ihren Visual Basic-Applikationen steuern können.

**Globales
Schlusswort**

6.12 Das Steuerelement »OptionButton«

Das Optionsfeld (OptionButton) dient der Auswahl optionaler Eigenschaften. Mehrere Optionsfelder im gleichen Container, beispielsweise in einem Rahmen, beeinflussen sich gegenseitig, d.h. es kann immer nur ein Optionsfeld angewählt sein. Ein angewähltes Optionsfeld enthält einen schwarzen Punkt. Auch dieses Feld ist unveränderlich.

OptionButtons (Optionsfelder) kommen immer dann zum Einsatz, wenn es darum geht, den Anwender aus einer Reihe von vorgegebenen Wahlmöglichkeiten eine oder auch mehrere Optionen auswählen zu lassen.

In einem solchen Fall handelt es sich jeweils um Fragen, die mit Ja bzw. Nein eindeutig zu beantworten sind und demnach über eine Schalterfunktion (1 und 0 bzw. *True* und *False*) dargestellt werden können.

Hierbei ist der Einsatz von *OptionButtons* nicht wahlfrei, sondern muss bezogen auf die jeweilige Situation erfolgen. Die Frage, die man sich in diesem Zusammenhang stellen muss, lautet:

- ▶ Darf der Anwender nur eine der vorgegebenen Optionen auswählen oder auch mehrere?

OptionButtons sind dann gefragt, wenn nur jeweils eine einzige Option aus einer Gruppe von möglichen Optionen ausgewählt werden darf, die Möglichkeiten sich also gegenseitig ausschließen.

Ein Beispiel hierfür ist die Wahl einer Schriftart in einer Textverarbeitung. Da ein Zeichen nicht gleichzeitig in verschiedenen Schriftarten dargestellt werden kann, ist man gezwungen, sich für eine der zur Verfügung stehenden Schriftarten zu entscheiden.

Frame (Rahmen) Im Zusammenhang mit *OptionButtons* kommt zumeist noch ein weiteres Steuerelement zum Einsatz, nämlich der Rahmen (*Frame*), den wir ja schon in vorangegangenen Kapiteln kennen gelernt haben. Solch ein Rahmen erlaubt es, mehrere untergeordnete Steuerelemente (in diesem Fall *OptionButtons*) in Gruppen zusammenzufassen.

Das zielt nicht nur darauf ab, darin übersichtlichere Dialogfelder zu erhalten, in denen die verschiedenen Funktionsgruppen rein optisch sauber voneinander abgrenzt sind. Vielmehr liegt die Hauptaufgabe eines Rahmens darin, Visual Basic darüber zu informieren, welche *OptionButtons* zu welcher Gruppe innerhalb eines Formulars gehören.

hierarchische Beziehung Im Gegensatz zu der Anordnung von Steuerelementen auf einem Formular, die keinen speziellen Regeln unterliegen und nur durch optische und funktionale Gesichtspunkte bestimmt werden, stehen in Rahmen angeordnete Steuerelemente und der Rahmen an sich in einer hierarchischen Beziehung zueinander.

Bei dieser Vorgehensweise nimmt der Rahmen die untergeordneten Steuerelemente praktisch auf, wodurch sie mit dem Rahmen (*Frame*) automatisch fest verknüpft werden.

untergeordnete Steuerelemente

Das ist deshalb notwendig, weil, wie bereits angesprochen, immer nur ein *OptionButton* einer Gruppe ausgewählt werden kann.

Sollen nun auf einer Form mehrere solcher Gruppen dargestellt werden, müssen diese funktional voneinander getrennt werden, damit in jeder Gruppe die Auswahl einer Option möglich ist und nicht nur eine Option für alle Gruppen.

Ein angenehmer Nebeneffekt ergibt sich dadurch auch in der Entwicklungsumgebung von Visual Basic, in welcher ein markierter Rahmen samt zugehöriger Steuerelemente bei der Formularerstellung frei verschiebbar ist, also als eine Einheit betrachtet wird.

6.12.1 Übung: Farben einstellen

Doch nun erst einmal genug der Theorie. Wenden wir uns nun unserer zu entwickelnden Übung zu.



Abbildung 6.90: Hintergrundfarbe einstellen

Um die Funktionalität von *OptionButtons* demonstrieren zu können, soll der Hintergrund des Steuerelements Bezeichnungsfeld (*Label*) durch Anklicken von Optionen entweder in der Farbe *Blau* oder *Grün* dargestellt werden (Abbildung 6.90).

Lösung

Hierzu benötigen Sie zu allererst das Steuerelement *Label* (Bezeichnungsfeld), das in unterschiedlichen Hintergrundfarben dargestellt werden soll.

Als Nächstes können Sie daran gehen, einen Rahmen auf der Form zu erstellen. Doppelklicken Sie hierzu auf das Symbol *Frame* in der Werkzeugleiste, worauf ein Rahmen in Standard-Größe auf der Form positioniert wird. Gestalten Sie danach Ihr Formular so, wie in Abbildung 6.90 ersichtlich, und geben Sie dem Rahmen die Überschrift (*Caption*) *Hintergrundfarbe*.

Rahmen (Frame) erstellen

Da sich Farben bei der Verwendung für die Darstellung von Zeichen bekanntermaßen gegenseitig ausschließen, werden *OptionButtons* (Optionsfelder) im Formular integriert.

Diese müssen also nun, wie bereits beschrieben, mit dem schon erstellten Rahmen verknüpft werden. In Visual Basic gibt es leider keine Auswahl, die mit *Frame verknüpfen* heißt – es muss also anders gehen.

OptionButtons hinzufügen

Um dem Rahmen *Frame1* einen *OptionButton* zuzuordnen, gibt es zwei Möglichkeiten:

- ▶ Doppelklicken Sie auf das Symbol *OptionButton* in der Werkzeugleiste, worauf ein *OptionButton* in Standard-Größe in der Form erstellt wird.



Nun reicht es aber nicht, das markierte Steuerelement *OptionButton* einfach per Drag&Drop in den Rahmen zu verschieben, da dadurch keinerlei Verbindung hergestellt wird. Stattdessen müssen Sie es über die Tastenkombination **[Shift]-[Entf]** ausschneiden, den Rahmen selektieren und anschließend das Steuerelement *OptionButton* über die Tastenkombination **[Shift]-[Einf]** im Rahmen (*Frame*) an der gewünschten Stelle platzieren.

- ▶ Klicken Sie auf das Symbol *OptionButton* in der Werkzeugleiste nur einmal, positionieren Sie den Mauszeiger an der gewünschten Stelle innerhalb des Rahmens und ziehen Sie das *OptionButton* in benötigter Größe auf.

Fügen Sie nun nach einer der beiden Möglichkeiten zwei *OptionButtons* in den Rahmen *Frame1* ein.

die Eigenschaft Value

Damit Sie als Programmierer programmintern feststellen können, ob der Anwender einen *OptionButton* ausgewählt (angeklickt) hat, um dann entsprechend auf diese Änderung des jeweiligen Status reagieren zu können, gibt es in Visual Basic die Eigenschaft *Value*.



Die Steuerelemente *OptionButton* und *CheckBox* verhalten sich, was die Rücklieferung des Ergebniswertes in der Eigenschaft *Value* betrifft, unterschiedlich.

Während bei einem *OptionButton*, der angeklickt wird, der Wert *True* (-1) und im anderen Falle der Wert *False* (0) über die *Value*-Eigenschaft rückgeliefert wird, erhalten Sie bei *CheckBoxen* den Wert 1 zurück, falls er mit einem *Häkchen* versehen (aktiviert) wurde, andernfalls den Wert 0.

Eigenschaft Value vorbelegen

Die Eigenschaft *Value* dient aber nicht nur dazu, die Auswahl des Anwenders beim Programmablauf festzustellen, sondern lässt sich auch über das Eigenschaftfenster vorbelegen.

Da bei der Verwendung von *OptionButtons* immer eine Option ausgewählt werden muss, belegt Visual Basic standardmäßig den ersten *OptionButton* einer Gruppe vor. Wollen Sie eine davon abweichende Vorbelegung, setzen Sie die Eigenschaft *Value* Ihres Standard-Optionsfeldes auf *True*.

Damit Sie nicht fälschlicherweise mehrere Optionen in einer Gruppe mit dem Wert *True* in der Eigenschaft *Value* vorbelegen, sorgt Visual Basic dafür, dass immer nur eine Option, und zwar die zuletzt aktivierte, die aktuelle Option ist und die anderen jeweils automatisch zurückgesetzt werden.

Hauchen wir nun dem Rahmen *Hintergrundfarbe* Leben ein. Doppelklicken Sie dazu auf den ersten *OptionButton* in diesem Rahmen, nämlich auf *Option1*.

Funktionalität integrieren

Hierauf öffnet Visual Basic die Ereignisprozedur *Option1_Click*, in der Sie nun folgende Anweisungen hinterlegen können:

```
If Option1.Value = True Then
    Label1.BackColor = &HFF0000
End If
```

In dieser Ereignisprozedur wird der *OptionButton* des Rahmens *Hintergrundfarbe* auf den Wert *True* hin abgefragt, also daraufhin, ob er der aktive *OptionButton* ist.

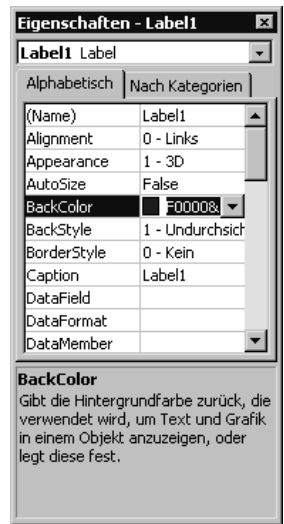


Abbildung 6.91: Farbwert holen

Den gewünschten Farbwert können Sie jederzeit ganz einfach über das Eigenschaftenfenster zum Steuerelement *Label* Eigenschaft *BackColor* holen, indem der hexadezimale Wert der über Palette ausgewählten Farbe in Ihren Programmcode kopiert wird (Abbildung 6.91).

Der Basic-Code der Ereignisprozedur des zweiten *OptionButtons* sieht genauso einfach aus:

```
Private Sub Option2_Click()
    If Option2.Value = True Then
        Label1.BackColor = &HFF00&
    End If
End Sub
```

Ist der zweite `OptionButton` der aktive, so wird die Vordergrundfarbe des Labels auf *Grün* gesetzt.

Starten Sie nun Ihre Applikation z.B. über `F5` und aktivieren den `OptionButton` *Blau*, so wird, wie in Abbildung 6.90 zu sehen, die Hintergrundfarbe im Label entsprechend dargestellt.

6.13 Das Steuerelement »CheckBox«

Das Kontrollkästchen (`CheckBox`) wird normalerweise dazu benutzt, dem Benutzer Gelegenheit zu geben, eine Eigenschaft ein- oder auszuschalten. Dabei wird im eingeschalteten Zustand ein Häkchen angezeigt.

Dem Kästchen kann ein Text zugeordnet werden. Mehrere Kontrollkästchen beeinflussen sich nicht gegenseitig, d.h. es können mehrere Kontrollkästchen mit einem Häkchen aktiviert werden. Sein Aussehen ist unveränderbar.

CheckBoxen (Kontrollkästchen) kommen immer dann zum Einsatz, wenn es darum geht, den Anwender aus einer Reihe von vorgegebenen Wahlmöglichkeiten eine oder auch mehrere Optionen auswählen zu lassen.

In einem solchen Fall handelt es sich jeweils um Fragen, die mit Ja bzw. Nein eindeutig zu beantworten sind und demnach über eine Schalterfunktion (1 und 0 bzw. *True* und *False*) dargestellt werden können.

Hierbei ist der Einsatz von *OptionButtons* und *CheckBoxen* nicht wahlfrei, sondern muss bezogen auf die jeweilige Situation erfolgen. *CheckBoxen* sind dann gefragt, wenn eine oder mehrere Optionen aus einer Gruppe von möglichen Optionen ausgewählt werden dürfen, die Möglichkeiten sich also gegenseitig nicht ausschließen.



Zum Beispiel schließen sich bei der Bestimmung von Formatierungen für ein Zeichen oder einen beliebigen Text *Fettdruck*, *Kursivschrift* und *Unterstreichen* keineswegs aus – sie können sogar beliebig kombiniert werden.

Dem muss dahingehend Rechnung getragen werden, dass der Anwender dann auch mehrere Auswahlen gleichzeitig treffen können muss, was dem klassischen Einsatz von *CheckBoxen* entspricht.

Frame (Rahmen)

Im Zusammenhang mit *CheckBoxen* kommt zumeist noch ein weiteres Steuerelement zum Einsatz, nämlich der Rahmen (*Frame*). Solch ein Rahmen erlaubt es, mehrere untergeordnete Steuerelemente (in diesem Fall *CheckBoxen*) in Gruppen zusammenzufassen.

Das zielt bei *CheckBoxen* im Gegensatz zu *OptionButtons* nur darauf ab, darin übersichtlichere Dialogfelder zu erhalten, in denen die verschiedenen Funktionsgruppen rein optisch sauber voneinander abgegrenzt sind.

6.13.1 Übung: Text formatieren

Doch nun erst einmal genug der Theorie. Wenden wir uns jetzt unserer zu entwickelnden Übung zu.



Abbildung 6.92:
Formatierung
einstellen

Um die Funktionalität von *CheckBoxen* demonstrieren zu können, soll der Text im Steuerelement Bezeichnungsfeld (*Label*) durch Anklicken von Optionen entweder *Fett*, *Kursiv* oder *Fett und Kursiv* formatiert werden (Abbildung 6.92).

Lösung

Hierzu benötigen Sie zu allererst das Steuerelement *Label* (Bezeichnungsfeld), das den Text »Visual Basic ist SUPER« in der Überschrift (Eigenschaft *Caption*) trägt und später dann unterschiedlich formatiert werden soll.

Als Nächstes können Sie daran gehen, einen Rahmen auf der Form zu erstellen. Doppelklicken Sie hierzu auf das Symbol *Frame* in der Werkzeugleiste, worauf ein Rahmen in Standard-Größe auf der Form positioniert wird. Gestalten Sie danach Ihr Formular so, wie in Abbildung 6.92 ersichtlich, und geben Sie dem Rahmen die Überschrift (*Caption*) *Formatierung*.

Für die Auswahl von Formatierungskriterien, die bei der Darstellung unseres Beispieltexes im Label *Label1* berücksichtigt werden sollen, scheidet *Option-Buttons* für die Lösung dieses Problemes automatisch aus, da es, wie bereits besprochen, durchaus möglich ist, mehrere Formatierungskriterien beliebig miteinander zu kombinieren.

Positionieren Sie deshalb innerhalb des Rahmens zwei *CheckBoxen* (Abbildung 6.92).

Bei der *CheckBox* mit dem Wert *Fett* in der Eigenschaft *Caption* könnte die Eigenschaft *Value* auf *1* gesetzt werden, wodurch das Kästchen bereits bei Programmstart mit einem *Häkchen* versehen würde.

Wie sieht nun die programmtechnische Realisierung bei *CheckBoxen* aus? Um festzustellen, ob der Anwender ein Häkchen in die *CheckBox* eingetragen oder entfernt hat, muss die Eigenschaft *Value* der jeweiligen *CheckBox* abgefragt werden.

**Rahmen (Frame)
erstellen**

**CheckBoxen
hinzufügen**

**Funktionalität
integrieren**

Der Programmcode für die Ereignisprozeduren *Check1_Click* und *Check2_Click* sieht deshalb so aus:

```
Private Sub Check1_Click()  
If Check1.Value = 1 Then  
    Label1.FontBold = True  
Else  
    Label1.FontBold = False  
End If  
If Check2.Value = 1 Then  
    Label1.FontItalic = True  
Else  
    Label1.FontItalic = False  
End If  
End Sub  
Private Sub Check2_Click()  
If Check2.Value = 1 Then  
    Label1.FontItalic = True  
Else  
    Label1.FontItalic = False  
End If  
If Check1.Value = 1 Then  
    Label1.FontBold = True  
Else  
    Label1.FontBold = False  
End If  
End Sub
```

Dabei muss, da im Gegensatz zu *OptionButtons* hier mehrere Auswahlen gleichzeitig möglich sein können, ein *Sonst (ELSE)*-Weg vorgesehen werden. Hat eine der Checkboxes kein Häkchen, sprich ist nicht ausgewählt, so muss auch die entsprechende Text-Formatierung im *Label* zurückgenommen werden.

Starten Sie nun Ihre Applikation z.B. über F5 und aktivieren die CheckBoxen *Fett* und *Kursiv*, so wird, wie in Abbildung 6.92 zu sehen, der Text im Label dementsprechend dargestellt.

6.14 Das Steuerelement »ComboBox«

Das Kombinationsfeld (ComboBox) vereint die Eigenschaften eines Textfeldes und eines Listenfeldes in sich. Dabei stehen drei Grundtypen zur Auswahl:

- ▶ nur Listenfeld,
- ▶ Listenfeld zum Aufklappen mit Anzeige der Selektion,
- ▶ und aufklappbares Listenfeld mit editierbarem Selektionsfeld sind verfügbar.

Das Aussehen ist durch Eigenschaften beeinflussbar.

ComboBoxen erleichtern den Anwendern immer dann die Arbeit, wenn mehrere gültige Möglichkeiten zur Wahl stehen. Mit ihrer Hilfe werden dem Anwender die relevanten Optionen in Listenform aufgeführt, aus denen er dann per Mausklick oder über die Cursortasten die gewünschte Option auswählen kann.

Da der Anwender eine gute Übersicht über alle verfügbaren Möglichkeiten erhält, muss er nicht mühsam verschiedene Varianten der Eingabe ausprobieren, nur weil er sich z. B. nicht mehr genau erinnern kann, wie ein bestimmter Begriff geschrieben wird oder er aus dem Kontext heraus nicht auf Anhieb versteht, welche Eingabe von ihm erwartet wird.

Die *ComboBox* kombiniert die Möglichkeiten einer Text- und einer *ListBox* (wird im nachfolgenden Kapitel vorgestellt). Das heißt, dass der Anwender sowohl Text eingeben als auch einen Eintrag aus der anhängenden Liste auswählen kann.

Der Hauptvorteil, den *ComboBoxen* gegenüber *ListBoxen* aufweisen können, ist der, dass sie extrem platzsparend sind, da die anhängende Liste mit den Wahlmöglichkeiten erst dann aufgeklappt wird, wenn der Anwender auf die zugehörige Schaltfläche mit dem nach unten weisenden Pfeil klickt.

Eingesetzt wird die *ComboBox* sinnvollerweise dann, wenn vom Anwender eine Eingabe erwartet wird, für die bestimmte Standards vorgegeben sind, wie z. B. die Angabe der Staatsangehörigkeit oder einer Anrede in einer Personalkartei.

Als Programmierer gibt man in solchen Fällen nur die gebräuchlichsten Auswahlmöglichkeiten vor, da es dem Anwender überlassen bleiben soll, Eingaben, die er nicht vorgegeben findet, dennoch über das Textfeld zu erfassen.

Diese Eingaben sollten dann sinnvollerweise gleich der Auswahlliste hinzugefügt werden, wodurch diese Liste flexibel bleibt und sich den individuellen Anforderungen des jeweiligen Anwenders automatisch anpasst.

6.14.1 Übung: Aus einer Artikelliste auswählen

In diesem Projekt soll es möglich werden, einen Eintrag (Artikel) aus einer Artikelliste auswählen und in einem Bezeichnungsfeld (*Label*) ausgeben zu können (Abbildung 6.93).

Alle möglichen Artikelbezeichnungen sollen automatisch bei Applikationsstart in die *ComboBox* alphabetisch sortiert gefüllt werden (Abbildung 6.94).

Lösung

Integrieren Sie, wie in den Abbildungen 6.93 und 6.94 zu sehen, in Ihrem Formular drei Bezeichnungsfelder (*Label*) und ein Kombinationsfeld (*ComboBox*).



Abbildung 6.93:
Eintrag (Artikel)
gewählt und
ausgegeben

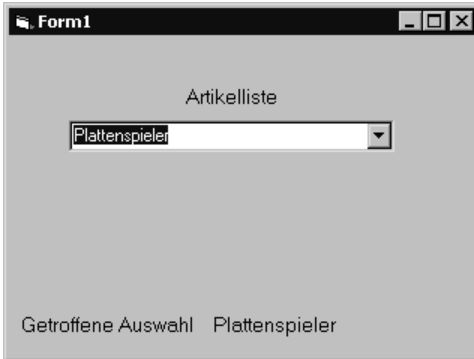


Abbildung 6.94:
Artikelliste alpha-
betisch sortiert in
die ComboBox
gefüllt



Einträge hinzufügen

Da die ComboBox automatisch bei Applikationsstart die Wahlmöglichkeiten anbieten soll, ist es nötig, die Box zu diesem Zeitpunkt zu füllen. Fügen Sie deshalb der Ereignisprozedur *Form_Load* die folgenden Zeilen hinzu (die Einträge müssen nicht alphabetisch sortiert sein):

```
Private Sub Form_Load()  
    Combo1.AddItem "Videorekorder"  
    Combo1.AddItem "Fernseher"  
    Combo1.AddItem "Stereolanlage"  
    Combo1.AddItem "CD-Player"  
    Combo1.AddItem "CD-Brenner"  
    Combo1.AddItem "Plattenspieler"  
End Sub
```



AddItem ist eine Methode, die zur Laufzeit ein Element in eine List- oder ComboBox einfügt.

Starten Sie nun das Programm und testen Sie die neue Funktionalität. Geschlossen belegt die *ComboBox* nur eine einzige Zeile. Klicken Sie in der ComboBox auf die Schaltfläche mit dem Pfeil, so klappt die ComboBox auf und zeigt Ihnen die definierten Auswahlen an (Abbildung 6.94).

Die durch die Prozedur *Form_Load* unsortiert geladenen Einträge werden nur deshalb in der ComboBox sortiert visualisiert, weil die Eigenschaft *Sorted* der ComboBox auf *True* gesetzt wurde.



Der Zugriff auf den aktuell angewählten Eintrag erfolgt über die Eigenschaft *List*. Die Nummer des Eintrags, also der Index, welcher Eintrag ausgewählt wurde, wird über die Eigenschaft *ListIndex* gesetzt oder abgefragt.

**Eintrag auswählen
und anzeigen**

Der erste Eintrag beginnt mit der Nummer *0*. Ist kein Eintrag angewählt, hat die Eigenschaft *ListIndex* den Wert *-1*.



So ist die Ereignis-Prozedur *Combo1_Click* wie folgt zu füllen:

```
Private Sub Combo1_Click()  
    Label3.Caption = Combo1.List(Combo1.ListIndex)  
End Sub
```

Dem Bezeichnungsfeld *Label3* ist der Eintrag in der Eigenschaft *Caption* zu übergeben, der in der ComboBox ausgewählt wurde.

Starten Sie nun Ihre Applikation z.B. über **F5** und wählen einen Eintrag durch Mausclick aus der ComboBox aus, so wird dieser zusätzlich im Bezeichnungsfeld visualisiert (Abbildung 6.93).

6.15 Das Steuerelement »ListBox«

Im Listenfeld (ListBox) kann sich der Benutzer eine Liste von Elementen anzeigen lassen, die auswählbar sind. Es werden automatisch Bildlaufleisten angezeigt. Sein Inhalt kann sortiert, sein Aussehen über Eigenschaften verändert werden.

Ein weiteres interessantes und mächtiges Steuerelement ist die *ListBox*, auch *Listenfeld* genannt. Es ermöglicht die Anzeige von Daten in Form einer Liste. Zu diesem Element gehört ein umfangreicher Satz an Eigenschaften und Methoden, die wir uns im Folgenden näher ansehen wollen.

ListBoxen erleichtern den Anwendern immer dann die Arbeit, wenn mehrere gültige Möglichkeiten zur Wahl stehen.

Mit ihrer Hilfe werden dem Anwender die relevanten Optionen in Listenform aufgeführt, aus denen er dann per Mausclick oder über die Cursortasten die gewünschte Option auswählen kann.

Da der Anwender eine gute Übersicht über alle verfügbaren Möglichkeiten erhält, muss er nicht mühsam verschiedene Varianten der Eingabe ausprobieren, nur weil er sich z.B. nicht mehr genau erinnern kann, wie ein bestimmter Begriff geschrieben wird, oder er aus dem Kontext heraus nicht auf Anhieb versteht, welche Eingabe von ihm erwartet wird.



Ein treffendes Beispiel für solch eine unterstützende Funktion ist der Dialog *Drucker* in Windows, in welchem Sie aus einer Druckerliste den oder die Drucker auswählen können, die Sie installieren möchten. In der Regel können Sie nur einen einzelnen Eintrag aus einer Liste wählen.

Es gibt jedoch auch einige Fälle, in denen sich mehrere Einträge gleichzeitig wählen lassen, wie z.B. im Verzeichnisenster des Explorers von Windows, der eine spezielle Art von *ListBox* darstellt.



Eine *ListBox* ist eine Liste mit maximal bis zu 32767 Einträgen, aus der mit Hilfe der Maus oder den Cursortasten ein Eintrag für die weitere Verarbeitung ausgewählt werden kann.

Im Gegensatz zur Standard-*ComboBox* ist eine *ListBox* stets vollständig, mit der zur Designzeit festgelegten Größe, sichtbar.



6.15.1 Übung: Musiktitel einer *ListBox* hinzufügen

In diesem Projekt, das außerdem die Basis für weitere Aufgabenstellungen bilden wird, soll es möglich werden, beliebige Musiktitel einiger Interpreten über eine *TextBox* erfassen und diese dann quasi auf »Knopfdruck« in eine *ListBox* übernehmen zu können (Abbildung 6.95).

Abbildung 6.95:
Musiktitel
hinzufügen

Lösung

Erstellen Sie dazu das Formular in der Visual Basic-Entwicklungsumgebung, wie in Abbildung 6.95 sichtbar, mit den Steuerelementen *Label*, *TextBox*, *CommandButton* und *ListBox*.

Einträge hinzufügen

Wir haben nun alle Steuerelemente, die für unser Beispiel notwendig sind, in der Form definiert. Jetzt muss der Befehlsschaltfläche (*CommandButton*) mit dem Namen *Command1* Leben eingehaucht werden, damit die in der *TextBox* erfassten Daten auch in die *ListBox* übernommen und angezeigt werden können.

Die Einträge in die Liste werden während des Programmlaufs mit Hilfe der Methode *AddItem* vorgenommen. Doppelklicken Sie also auf den *Command-Button*, worauf die Ereignisprozedur *Command1_Click ()* geöffnet wird, und geben Sie folgenden Programmcode ein:

```
Private Sub Command1_Click()  
    List1.AddItem Text1.Text  
End Sub
```

Zusätzlich könnten Sie auch die Position jedes neuen Eintrags in der Liste bestimmen, deren oberster Eintrag den Index *0* aufweist. Wird diese Position nicht explizit angegeben, so wird der Eintrag immer an das Ende der *ListBox* angehängt.

Im folgenden Beispiel würde in die *ListBox* ein Eintrag an fünfter Stelle eingefügt:

```
List1.AddItem Text1.Text, 4
```

Beinhaltet eine *ListBox* mehr Einträge, als gleichzeitig angezeigt werden können, so versieht Visual Basic das Steuerelement automatisch mit *ScrollBars* (Bildlaufleisten), mit deren Hilfe Sie sich innerhalb der *ListBox* bewegen können.

Starten Sie nun das Programm zum Test, und geben Sie einige Musiktitel ein, um die korrekte Übernahme der eingegebenen Daten in die *ListBox* kontrollieren zu können (Abbildung 6.95).

6.15.2 Übung: Musiktitel aus der *ListBox* auslesen

Als Nächstes soll es möglich werden, einen in der *ListBox* mit der Maus angewählten Eintrag (angeklickt) auszulesen und in der *Textbox* auszugeben (Abbildung 6.96).

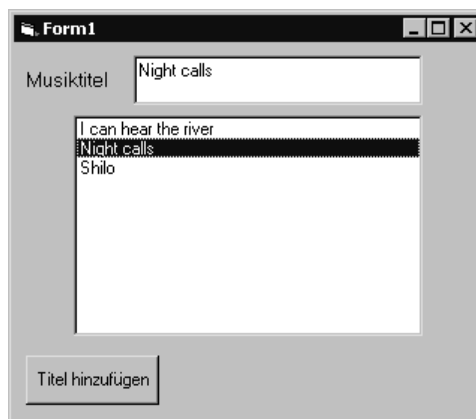


Abbildung 6.96:
Markierten *ListBox*-
eintrag auslesen

Lösung

Eintrag auslesen

Mit der Eigenschaft *ListIndex* ermitteln Sie den Index des aktuell markierten Satzes (Eintrags) einer ListBox. Diese Information können Sie dann anschließend dazu verwenden, um den Datensatz (Eintrag) auszulesen, zu bearbeiten oder auch zu löschen.

Soll z.B. der markierte Eintrag aus einer ListBox ausgelesen werden, um ihn weiterzuverarbeiten, dann benötigen Sie zusätzlich die Eigenschaft *List*, die wiederum die Information über den Index des Eintrags benötigt, der mit der Eigenschaft *ListIndex* ermittelt wird.

Um diese Eigenschaften auszutesten, soll ja in unserer Übung der Inhalt des in der Listbox markierten Musiktitels in die Textbox übernommen werden. Tragen Sie hierzu in der Ereignis-Prozedur *List1_Click* die folgenden Codezeilen ein:

```
Private Sub List1_Click()  
    Text1.Text = List1.List(List1.ListIndex)  
End Sub
```

Starten Sie nun das Programm und überzeugen Sie sich vom korrekten Ablauf dieser Funktion, indem Sie wieder einige Musiktitel erfassen und diese dann anschließend per Mausklick in der Listbox markieren (Abbildung 6.96).



6.15.3 Übung: Musiktitel aus der Listbox löschen

Jetzt soll es möglich werden, einen in der Listbox mit der Maus angewählten Eintrag (angeklickt) über eine weitere Schaltfläche aus der Listbox löschen zu können (Abbildung 6.97).

Abbildung 6.97:
Markierten List-
boxeintrag löschen



Lösung

Erstellen Sie dazu das Formular in der Visual Basic-Entwicklungsumgebung mit einem weiteren Steuerelement *CommandButton* (wie in Abbildung 6.97 sichtbar).

Mit der Methode *RemoveItem* entfernt man einen beliebigen Eintrag aus einer *ListBox*. Natürlich sollte dieses Löschen nicht dem Zufall überlassen werden, sondern gezielt erfolgen. Darum benötigt auch diese Methode einen Index, der auf den jeweils markierten Satz (Eintrag) verweist.

Nachdem Sie die Oberfläche, wie in der Abbildung 6.97 zu sehen, um einen *CommandButton* erweitert haben, können Sie nun die folgende Zeile in die Ereignis-Prozedur *Command2_Click ()* einfügen:

```
Private Sub Command2_Click()  
    List1.RemoveItem (List1.ListIndex)  
End Sub
```

Starten Sie nun das Programm und überzeugen Sie sich vom korrekten Ablauf dieser neuen Funktion.

Eintrag löschen**6.15.4 Übung: Alle Musiktitel aus der ListBox löschen**

Über eine weitere Schaltfläche soll es möglich werden, alle Musiktitel, sprich alle Einträge aus der *ListBox* zu entfernen, also die *ListBox* komplett zu leeren (Abbildung 6.98).



Abbildung 6.98:
Alle Listboxein-
träge entfernen

Lösung

Erstellen Sie dazu das Formular in der Visual Basic-Entwicklungsumgebung mit einem weiteren Steuerelement *CommandButton* (wie in Abbildung 6.98 sichtbar).

alle Einträge entfernen

Soll nicht nur ein einzelner Eintrag einer ListBox gelöscht werden, sondern deren gesamter Inhalt, wird die Methode *Clear* eingesetzt, die nur den Namen der jeweiligen ListBox als Parameter benötigt.

Nachdem Sie die Oberfläche, wie in der Abbildung 6.98 zu sehen, um einen *CommandButton* erweitert haben, können Sie nun die folgende Zeile in die Ereignis-Prozedur *Command3_Click ()* einfügen:

```
Private Sub Command3_Click()
    List1.Clear
End Sub
```

Starten Sie nun das Programm und überzeugen Sie sich vom korrekten Ablauf dieser neuen Funktion.



6.15.5 Übung: Musiktitel sortieren und multiselektierbar machen

In dieser letzten Übung soll es ermöglicht werden, Musiktitel automatisch in der ListBox bei Titelerfassung alphabetisch richtig einzusortieren. Außerdem sollen nicht nur ein Eintrag, sondern auch mehrere Listboxeinträge angewählt werden können (Abbildung 6.99).

Abbildung 6.99:
Multiselektierbar
und sortiert



Lösung

ListBox-Inhalt sortieren

Mit dem Setzen der Eigenschaft *Sorted* auf *True* kann der komplette Inhalt der ListBox alphabetisch aufsteigend sortiert werden.

Wenn in unserem Beispiel die Liste mit *Sorted=True* sortiert wird, wird automatisch der neue über die Methode *AddItem* hinzugefügte Eintrag an der richtigen Position innerhalb der Sortierreihenfolge angelegt (Abbildung 6.99).



Grundsätzlich erfolgt die Sortierung zeichenorientiert. Dies kann dazu führen, dass, wenn Zahlen am Anfang eines Eintrags stehen, Sie möglicherweise nicht die erwünschte Sortierung bekommen. Die Reihenfolge wird vom Zeichenwert (ASCII/ANSI-Wert) und nicht vom numerischen Wert bestimmt.

Zum Beispiel würden die Einträge 0, 1, 10, 11, 12, 19, 2, 20, 21, 29, 3, 30, 31, 32 etc. nicht richtig sortiert angezeigt werden. Die Sortierung zeigt allerdings dann das gewünschte Ergebnis, wenn Sie eine konstante Anzahl von Ziffern verwenden, also für die einstelligen Ziffern eine Null voranstellen, zum Beispiel 01, 02, 03, ... 10, 11, 12, ... 20, 21 etc.



Wenn Sie die Eigenschaft *MultiSelect* auf *Einfach* setzen, wird jeder Eintrag, den Sie anklicken, in der Liste selektiert. Erst ein erneuter Klick innerhalb der ListBox entfernt die Selektion. Beim Wert *Erweitert* können Sie durch Ziehen mit der Maus und gedrückter linker Maustaste ganze Bereiche selektieren.

**mehrere Einträge
anwählen**

Durch Klicken mit gedrückter **[Strg]**-Taste werden einzelne Einträge selektiert (Abbildung 6.99). Klicken mit gedrückter **[Shift]**-Taste selektiert einen Bereich vom zuletzt selektierten bis zum aktuellen Eintrag. Diese Erweiterungen können beliebig miteinander kombiniert werden.

Mit dem Setzen der Eigenschaft *Style* auf *Kontrollkästchen* wird jedem Listeneintrag ein *Kontrollkästchen* (*CheckBox*) vorangestellt, das Sie durch Klick aktivieren (Häkchen) oder deaktivieren (leer) können.

**Kontrollkästchen
voranstellen**

6.16 Das Steuerelement »ScrollBars«

Bildlaufleisten (horizontal und vertikal) ermöglichen die grafische Anzeige und Eingabe einer Position in einem Feld, dessen Inhalt sich nicht vollständig im Fenster darstellen lässt. Die Berechnung der Position muss manuell im Code erfolgen. Ihr Aussehen ist durch Eigenschaften sehr begrenzt zu beeinflussen.

6.16.1 Übung: DM-Betrag über horizontale Bildlaufleiste wählen

Über eine horizontale Bildlaufleiste *HScrollBar* soll ein DM-Betrag in ganzen DM zwischen 0 und 2000 in 10er-Schritten durch Klicken fließend ausgewählt und in einem Bezeichnungsfeld (*Label*) visualisiert werden (Abbildung 6.100).

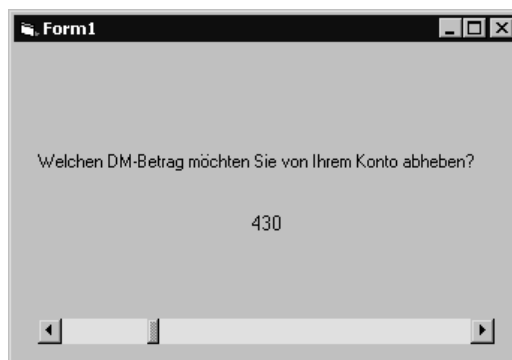
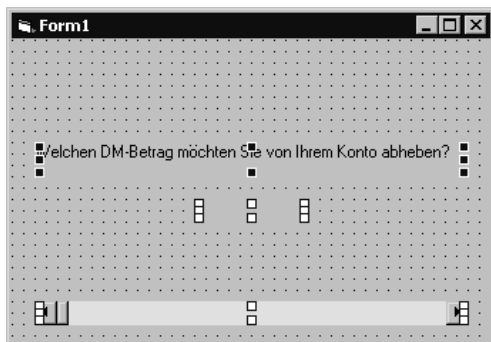


Abbildung 6.100:
Horizontale Scroll-
Bar im Einsatz

Lösung

Abbildung 6.101:
Zwei Label und
eine HScrollBar im
Formular
aufgenommen



Gestalten Sie Ihr Formular so, wie in Abbildung 6.101 ersichtlich.

Damit bei Mausklick auf die horizontale Bildlaufleiste *Hscroll1* auch eine Zahl im Bezeichnungsfeld im Bereich von 0 bis 2000 erscheint, sind zuerst, bevor eine Ereignisprozedur programmiert werden kann, zwei für Bildlaufleisten elementar wichtige Eigenschaften zu setzen.

**Die Eigenschaften
Min und Max**

Da zwischen 0 und 2000 visualisiert werden soll, ist zum einen die Eigenschaft *Min* auf 0, sprich auf den minimalsten Wert, und zum anderen die Eigenschaft *Max* auf 2000, sprich auf den maximal anzuzeigenden Wert zu setzen.

Jetzt ist es uns möglich, die Ereignisprozedur, die auf das Verändern des ScrollBar-Anzeigers reagiert, zu erweitern:

```
Private Sub HScroll1_Change()  
    Label2.Caption = HScroll1.Value  
End Sub
```

Dem Bezeichnungsfeld *Label2* ist der Wert (*Value*) des ScrollBar-Anzeigers in der Eigenschaft *Caption* (Inschrift, Überschrift) zu übergeben.

**Die Eigenschaften
LargeChange und
SmallChange**

Da aber die Anforderung besteht, den DM-Betrag im Bereich von 0 und 2000 nicht mit einer Schrittweite von 1 (z.B. 1, 2, 3 usw.), sondern mit einer Schrittweite von 10 zu visualisieren, müssen Sie außer den Eigenschaften *Min* (*Min* = 0) und *Max* (*Max* = 2000) auch die Eigenschaften *LargeChange* (größtmögliche Schrittweite) und *SmallChange* (kleinstmögliche Schrittweite) anpassen (Abbildung 6.102).

Starten Sie nun Ihre Applikation z.B. über F5 und klicken auf die Pfeiltasten der horizontalen Bildlaufleiste, so kann, wie in Abbildung 6.100 zu sehen, ein DM-Betrag zwischen 0 und 2000 in der Schrittweite 10 ausgewählt und in einem Label angezeigt werden.

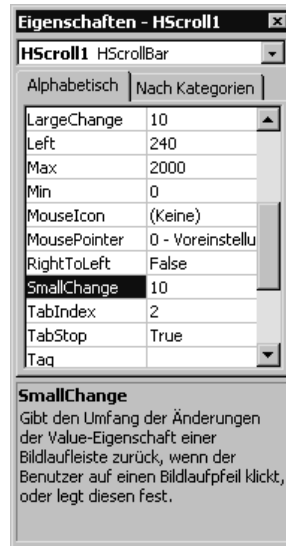


Abbildung 6.102: Eigenschaften LargeChange und SmallChange auf den Wert 10 gesetzt

6.17 Das Steuerelement »PictureBox«

Das Bildfeld (PictureBox) kann eine Grafik in den Formaten .bmp, .pcx, .gif, .jpg, .cur, .ico und Metafile anzeigen. Es passt sich entweder der Bildgröße an oder zeigt nur einen Ausschnitt des Bildes.

6.17.1 Übung: Auf Anforderung ein Bild laden und anzeigen

Über eine Befehlsschaltfläche (CommandButton) soll die Bitmap-Datei Setup.Bmp aus dem Ordner Windows der Festplatte C: in die PictureBox geladen werden.

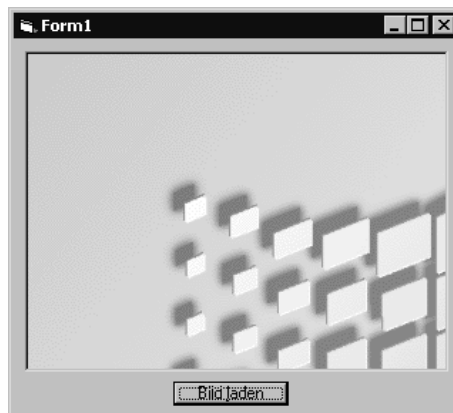


Abbildung 6.103: Geladenes Bild Setup.Bmp

Lösung

Damit bei Mausklick auf den Button auch das gewünschte Bild in der PictureBox erscheint, ist die Ereignisprozedur *Click* des CommandButtons zu füttern.

```
Private Sub Command1_Click()  
    Picture1.Picture = LoadPicture("C:\windows\setup.bmp")  
End Sub
```

Der Eigenschaft *Picture* des Bildfeldes ist der genaue Standort (*C:\windows\setup.bmp*) des anzuzeigenden Bildes zu übergeben. Die Funktion *LoadPicture* veranlasst das entsprechende Laden.

Starten Sie nun Ihre Applikation z.B. über **F5** und aktivieren den Button *Bild laden*, so erscheint, wie in Abbildung 6.103 zu sehen, die Setup-Bitmap in der PictureBox.



6.17.2 Übung: Bild im Formular zur Laufzeit frei bewegen

Sehr oft besteht in Applikationen die Anforderung, ein geladenes Bild anwenderindividuell an eine beliebige andere Position im Formular verschieben zu können. So soll, wie in den beiden Abbildungen 6.104 und 6.105 zu sehen, das Bild automatisch an die Stelle im Formular hinbewegt werden, auf die mit der Maus geklickt wurde.

Abbildung 6.104:
Bild an beliebiger
Position im
Formular

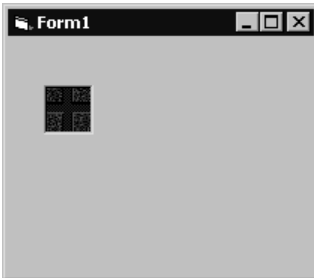
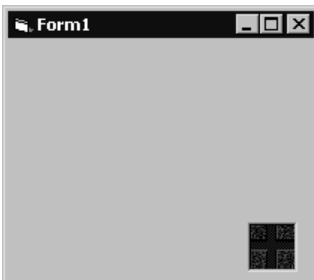


Abbildung 6.105:
Bild an andere
Position im
Formular bewegt



Lösung

Diese Übung haben wir u.a. aus diesem Grunde ausgewählt, weil hier wieder auf sehr schöne Art und Weise deutlich wird, mit wie wenig Aufwand (Programmzeilen, Anweisungen etc.) in Visual Basic sehr viel »bewegt« werden kann.

```
Private Sub Form_MouseDown(Button As Integer, Shift As Integer, X As
Single, Y As Single)
    Picture1.Move X, Y
End Sub
```

Da das Bild-Verschieben dann erfolgen soll, wenn die Maustaste im Formular gedrückt wird, muss dies irgendwie abgefragt werden können. Dazu stellt uns das Formular die Ereignisprozedur *MouseDown* mit den Mauskoordinaten (den Übergabeparametern *X* und *Y*) zur Verfügung.

Mit der Methode *Move* ist dann nur noch das Steuerelement *PictureBox* an die Stelle, bezogen auf die horizontalen (*X*) und die vertikalen Koordinaten (*Y*), zu verschieben.

6.18 Das Steuerelement »Timer«

Der Zeitgeber (*Timer*) kann periodisch Ereignisse auslösen. Die Abstände sind in Grenzen variabel. Das Aussehen ist durch Eigenschaften nicht beeinflussbar, da keine interaktive, grafische Schnittstelle vorliegt.

6.18.1 Übung: Uhrzeit anzeigen

In einem Bezeichnungsfeld (*Label*) soll die aktuelle Uhrzeit im Sekundentakt in Form einer digitalen Anzeige erscheinen (Abbildung 6.106).

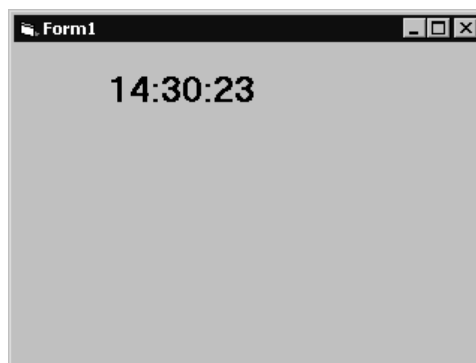


Abbildung 6.106:
Laufende Uhr in der
Applikation

Lösung

Integrieren Sie in Ihr Formular die Steuerelemente *Label* und *Timer*.

Damit sich der Zeitgeber automatisch aktiviert, muss diesem zuerst, bevor überhaupt eine Ereignisprozedur programmiert werden kann, mitgeteilt werden, in welchem Intervall er sich zu aktivieren hat.

Dazu ist die Eigenschaft *Interval* des Steuerelements *Timer1* zu modifizieren. Da im Sekundentakt die Uhrzeit anzuzeigen ist, muss *Interval* auf 1000 gesetzt werden.



Mit *Interval* legen Sie ein Intervall in Millisekunden fest, das aktiviert wird, wenn die Zeit vergangen ist. Dabei entsprechen beispielsweise 1000 Millisekunden 1 Sekunde. Der maximal einzustellende Wert beträgt 65535 Millisekunden, die eine Zeitspanne von etwas mehr als einer Minute darstellen. Hat *Interval* den Wert 0 (Voreinstellung), so ist das Zeitgeber-Steuerelement deaktiviert.

Jetzt ist es uns möglich, die Ereignisprozedur zu programmieren, die die Uhrzeit im Bezeichnungsfeld anzeigt.

```
Private Sub Timer1_Timer()  
    Label1.Caption = Time  
End Sub
```

Das Event *Timer* des Zeitgeber-Steuerelements *Timer1* wird in der von uns in der Eigenschaft *Interval* festgelegten Zeitspanne, sprich alle 1000 Millisekunden (also im Sekundentakt), aufgerufen.

Dem Bezeichnungsfeld *Label1* ist dann nur noch die Uhrzeit über die Funktion *Time* in der Eigenschaft *Caption* (Inschrift, Überschrift) zu übergeben.



Time ist eine Funktion zur Abfrage der aktuellen Uhrzeit des Betriebssystems.

Starten Sie nun Ihre Applikation, so erscheint, wie in Abbildung 6.106 zu sehen, die aktuelle Uhrzeit sekundlich im Bezeichnungsfeld (*Label*).



6.18.2 Übung: Entwicklung eines Terminplaners

Sicher haben Sie schon einmal unter Windows ein Programm gesucht, das Sie an mögliche Termine erinnern kann. Nun gibt es schon solche Programme, aber meistens handelt es sich um komplexe Terminkalender mit einer umfangreichen Funktionalität.

Was fehlt, ist ein einfaches Programm, das lediglich die Aufgabe erfüllt, eine einfache Liste von Terminen zu verwalten und abzuarbeiten (Abbildung 6.107). Die Übung ist also, ein solch einfaches Werkzeug zu erstellen.



Abbildung 6.107:
Definieren eines
Termins im
Terminplaner

Es ist immer ratsam, sich im Vorfeld Gedanken über den Funktionsumfang zu machen. Dabei stellt man sich zuerst die Fragen, welche Funktionen das Programm beinhalten soll, wie es optisch aussehen soll und welche Einschränkungen gelten sollen. Für die Übung des Terminplaners könnte man zu folgendem Ergebnis kommen:

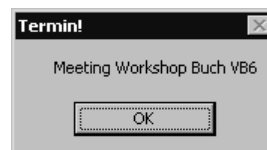


Abbildung 6.108:
Terminerinnerung

- ▶ Einfache, übersichtliche Bedienung (Abbildung 6.107).
- ▶ Keine Sekundenabfrage, daher auch keine Sekundeneingabe für die Termine.
- ▶ Genauigkeit der internen Zeitabfrage höchstens 5 Sekunden.
- ▶ Termine sollen gespeichert und nach einem Neustart wieder geladen werden.
- ▶ Terminen soll ein Kommentar zugeordnet werden können.
- ▶ Abgelaufene Termine werden gelöscht, der Benutzer wird darüber informiert.
- ▶ Akustisches Signal ist bei Termin aktivierbar, gilt dann für alle Termine (Abbildung 6.108).
- ▶ Keine übermäßige Prozessorbelastung durch stetige Abfragen.

Aufgabe analysieren

Daraus ergibt sich ein erstes Bild der Anwendung. Wir benötigen ein Formular für die Aufnahme der Steuerelemente und ein Codemodul für etwaige Funktionen und Prozeduren.

Da die Termine gespeichert werden sollen, benötigen wir Funktionen für das Speichern und Auslesen der Termindaten.

Hier bieten sich Funktionen aus dem Windows-API an, welche die Möglichkeit bieten, in einer Konfigurationsdatei unter einem Stichpunkt zusammengefasst, geordnet Daten abzulegen und wieder abzufragen.

Des Weiteren soll die Prozessorzeit nicht zu sehr in Anspruch genommen werden, da das Programm im Hintergrund arbeiten soll.

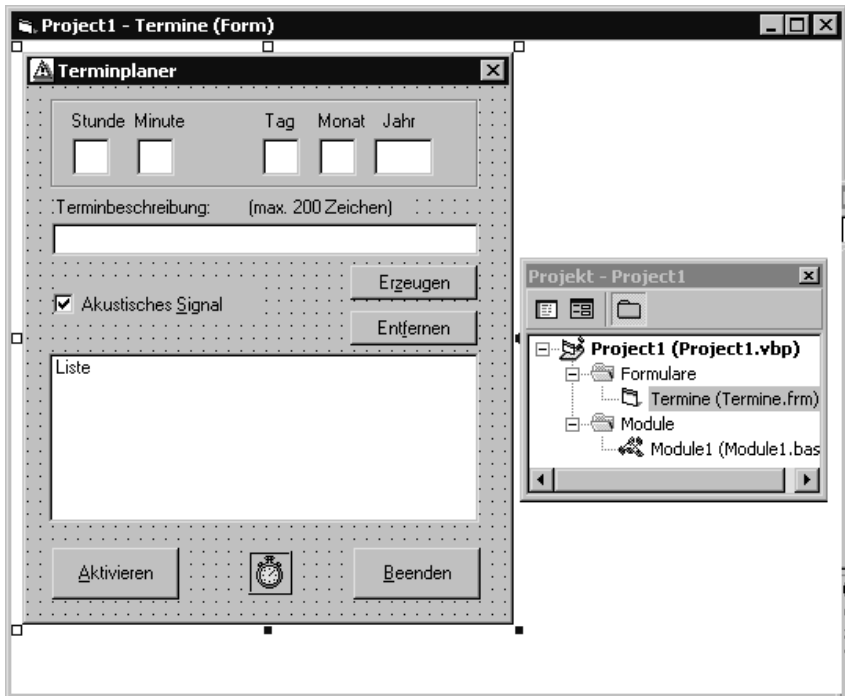
Daher kann keine Schleife verwendet werden, in der bei jedem Durchlauf auf einen Termin hin geprüft wird. Da die Genauigkeit nur ca. 5 Sekunden betragen soll, kann man die Prüfung der Termine in einem Timer-Ereignis des Steuerelements *Zeitgeber* vornehmen.

Dies wird nur alle fünf Sekunden kurz ausgelöst und verbraucht auch nur dann Prozessorleistung, die restliche Zeit steht anderen Programmen zur Verfügung.

Lösung

Projekt erstellen

Abbildung 6.109:
Das Formular und
der Projektexplorer



Erstellen Sie also ein neues Projekt. Visual Basic stellt Ihnen bereits ein Formular zur Verfügung. Dort können Sie nun Ihre Steuerelemente platzieren. Erzeugen Sie zusätzlich ein Codemodul (BAS-Modul) (Abbildung 6.109), das Ihre Funktionen und Prozeduren aufnehmen kann. Versuchen Sie das Formular, wie in Abbildung 6.109 ersichtlich, zu erstellen.

Die oberen Felder sind Textfelder. Ihre Länge ist auf 2 bzw. beim Jahr auf 4 Zeichen beschränkt (Eigenschaft *MaxLength*). Darunter befindet sich ebenfalls ein Textfeld, in das der Benutzer einen Kommentar zum jeweiligen Termin eingeben kann.

Die Befehlsschaltfläche ERZEUGEN erstellt einen Termin aus den obigen Angaben in der Listbox, die als Übersicht dient. Die Befehlsschaltfläche ENTFERNEN soll den in der Liste angewählten Termin wieder entfernen.

Die Liste selbst gibt eine Übersicht über alle aktiven Termine. Sie dient gleichzeitig der Verwaltung, also der Speicherung und Darstellung der Termine. Die Befehlsschaltfläche AKTIVIEREN soll die Terminüberprüfung im Hintergrund aktivieren.

Zu diesem Zweck ist es angebracht, das Programm auf ein Symbol zu minimieren, damit es auf der Arbeitsoberfläche nicht unnötig Platz belegt.

Die Befehlsschaltfläche BEENDEN schließt die Anwendung und speichert die noch unerledigten Termine ab. Das Codemodul wird benötigt, da die Deklaration externer Funktionen nicht im Deklarationsteil eines Formulars erfolgen darf. Außerdem können hier vorzugsweise eigene Prozeduren abgelegt werden.

Beginnen wir, die Objekte mit Code zu füllen. Fangen wir dabei oben an und beschreiben ein Textfeld. Wir müssen verhindern, dass der Benutzer Zeichen eingibt, die nicht zu einem Datum oder einer Uhrzeit gehören. Außerdem muss die Eingabe auf bestimmte Grenzen hin überwacht werden, damit nicht als Datum der 44.16.1781 eingegeben werden kann.

Hierfür werden zwei Ereignisse benötigt. *KeyPress*, um eine Eingabe illegaler Zeichen abzufangen, und *Change*, um die Einhaltung der Grenzen zu überwachen:

```
Private Sub Stunden_Change()  
    If Val(Stunden.Text) > 23 Then Stunden.Text = "23"  
End Sub  
Private Sub Stunden_KeyPress(KeyAscii As Integer)  
    Select Case KeyAscii  
        Case 8, 48 To 57      'Backspace(8) und von "0"(48) bis "9"  
                                zugelassen  
            'Nichts passiert  
        Case Else  
            KeyAscii = 0      'Keine Sonderzeichen und Buchstaben
```

Übersicht

nach dem Design folgt die Programmierarbeit

```
zulassen
    End Select
End Sub
```

In der Ereignisprozedur *LostFocus* kann schließlich beim Verlassen des Textfeldes der Inhalt in ein einheitliches Format gebracht werden. Wenn ein Benutzer nur ein Zeichen eingibt, wird ein zweites automatisch ergänzt; aus einer 1 wird eine 01. Der Befehl *Format* leistet hier gute Dienste:

```
Private Sub Stunden_LostFocus()
    Stunden.Text = Format$(Val(Stunden.Text), "00")
End Sub
```

Die Eingabelänge für Text im Kommentartextfeld muss per Eigenschaft *MaxLength* auf 200 Zeichen beschränkt werden. Ansonsten gelten keine weiteren Beschränkungen.

Termin eintragen

Die Befehlsschaltfläche ERZEUGEN soll nun einen Termin in die Liste eintragen. Zuerst muss daraufhin geprüft werden, ob überhaupt etwas in die Textfelder eingetragen wurde. Dies überprüfen Sie am Anfang der Ereignisprozedur für jedes Textfeld. Beispielhaft folgt die Prüfung des Minuten-Textfeldes:

```
Private Sub Erzeugen_Click()
Dim Eintrag As String
    If Minuten.Text = "" Then
        MsgBox "Eingabe unvollständig"
        Minuten.SetFocus
        Exit Sub
    End If
```

Die Methode *SetFocus* setzt die Eingabemarke auf das jeweilige Textfeld. Danach muss die Ereignisprozedur mit *Exit Sub* verlassen werden.

Liste erzeugen

Wurden alle Werte korrekt eingegeben, kann der Eintrag für die Liste erzeugt werden:

```
Eintrag$ = Stunden.Text & " : " & Minuten.Text & " "
Eintrag$ = Eintrag$ & Tage.Text & "." & Monate.Text & "." &
Jahre.Text
Eintrag$ = Format$(Eintrag$, "dd/mm/yyyy hh:nn") 'Formatieren
Eintrag$ = Eintrag$ & Space(5) & TerminText.Text 'Kommentar dazu
Liste.AddItem Eintrag
If Liste.List(Liste.NewIndex + 1) = Eintrag Then
    MsgBox "Eintrag bereits vorhanden", vbOKOnly, "Achtung!"
    Liste.RemoveItem Liste.NewIndex 'Eintrag wieder entfernen
    Exit Sub
End If
End Sub
```

Die Inhalte der Textfelder, einschließlich der des Kommentarfeldes, werden in der Zeichenkette *Eintrag\$* zusammengefügt. Die Anweisung *Format* hilft hier wiederum, den Termin in einem definierten Format zu erzeugen. Mit der Methode *AddItem* wird der Termin der Liste (*ListBox*) hinzugefügt.

Die Liste muss sortiert werden (Eigenschaft *sorted = True*). Sollte der gleiche Termin bereits vorhanden sein, wird der neue Eintrag vom Steuerelement eine Zeile oberhalb erzeugt. Daher kann mit der Abfrage dieses Eintrags ein doppelt vorhandener Termin gefunden und entfernt werden.

Die Schaltfläche AKTIVIEREN muss nun den Timer starten und die Termine abspeichern:

Timer starten

```
Private Sub Aktivieren_Click()
    If Liste.ListCount > 0 Then 'Wenn Termine vorhanden sind
        Me.WindowState = 1      'Als Symbol verkleinern
        Timer1.Interval = 5000 'Alle 5 Sekunden Termine überprüfen
        Timer1.Enabled = True  'Timer einschalten
        Schreibe_neue_Liste    'Liste abspeichern
    Else
        MsgBox "Sie haben keine Termine eingetragen", vbOKOnly,
        "Achtung"
    End If
End Sub
```

Alle 5 Sekunden wird der Timer aufgerufen, wobei dann die Termine untersucht werden müssen:

```
Private Sub Timer1_Timer()
    Dim jetzt As String, i As Integer, teststr As String
    jetzt$ = Format$(Now, "dd/mm/yyyy hh:nn")
    For i = 0 To Liste.ListCount - 1 'Für alle Elemente der
    Liste
        teststr$ = Left$(Liste.List(i), 18)
        If teststr$ = jetzt$ Then
            Me.WindowState = 0 'Fenster wieder in
            Normalgröße
            teststr$ = Mid$(Liste.List(i), 19, Len(Liste.List(i)) - 18)
            MsgBox teststr$, vbOKOnly, "Termin!" 'Nachricht
            Liste.RemoveItem i 'Termin entfernen
            Schreibe_neue_Liste 'Neue Liste abspeichern
            Liste.Clear 'aktuelle Liste löschen
            Liste_füllen 'Liste wieder beschreiben
            If Check_Beep.Value = True Then 'Wenn akustisches Signal
                Beep 'Dann zweimal piepen
                Beep
            End If
        End If
    Next i
End Sub
```

Now stellt die aktuelle Systemzeit mit Datum Ihres Rechners zur Verfügung. Es ist ein *Variant*-Datentyp vom Untertyp *Date*. Durch Wandlung in *Double* erhalten Sie einen numerischen Wert. Die Stellen vor dem Komma sind das Datum in Tagen seit 31.12.1899, die Nachkommastellen bilden die Uhrzeit bezogen auf 24 Stunden.

Das *Format* bedeutet Folgendes: zweistelliger Tag und Monat, vierstelliges Jahr getrennt durch die länderspezifischen Trennzeichen der Systemsteuerung. Dann drei Leerstellen und schließlich jeweils zweistellig Stunde und Minute, ebenfalls bezüglich der Systemsteuerung.

Windows API ansteuern

Die Deklaration der Funktionen zum Speichern und Lesen aus dem Windows API muss im allgemeinen Deklarationsabschnitt des Codemoduls erfolgen:

```
Declare Function WritePrivateProfileString Lib "kernel32" Alias _
    "WritePrivateProfileStringA" (ByVal Sektion As String, ByVal
    Identifier _
    As Any, ByVal Eintrag As Any, ByVal Dateiname As String) As Long
Declare Function GetPrivateProfileString Lib "kernel32" Alias _
    "GetPrivateProfileStringA" (ByVal Sektion As String, ByVal Identifier
    _
    As Any, ByVal Defaultwert As String, ByVal buffer As String, ByVal _
    Buf_Size As Long, ByVal Dateiname As String) As Long
```

WritePrivateProfileString schreibt den *Eintrag* in die *Sektion*, wo er im *Identifier* abgelegt wird. In der Datei *Dateiname* sieht das dann grundsätzlich wie folgt aus:

```
[Sektion]
Identifier=Eintrag
```

Zum Lesen benötigt die Funktion *GetPrivateProfileString* die Angaben zu *Sektion*, *Identifier* und einen *Buffer*, in welchen der Eintrag geschrieben werden kann. Dabei muss der *Buffer* ein String mit der festen Länge *Buf_Size* sein. Über *Defaultwert* kann ein String zurückgegeben werden, wenn der *Identifier* nicht gefunden wurde oder leer ist. Der Rückgabewert enthält die Anzahl der gelesenen Zeichen.

Mit diesen Voraussetzungen lassen sich folgende Prozeduren erstellen:

```
Public Sub Schreibe_neue_Liste()
    Dim i As Integer, lang As Long
    lang = WritePrivateProfileString("Termine", 0&, "", "termine.ini")
    'löschen
    For i = 0 To Termine.Liste.ListCount - 1      'für alle Termine in
    Liste
        lang = WritePrivateProfileString("Termine", Trim$(Str$(i)), Termine_
        .Liste.List(i), "termine.ini")          'Alles neu schreiben
    Next i
End Sub
```

Diese Prozedur schreibt alle Termineinträge der Liste in die Datei »Termine.ini« im *Windows*-Verzeichnis, wo sie gegebenenfalls automatisch erzeugt wird. Die Einträge werden nummeriert nach ihrem *ListIndex* abgelegt.

Termineinträge in Datei schreiben

```
Public Sub Liste_füllen()
Dim i As Integer, buffer As String * 255, lang As Long
lang = 1
Termine.Liste.Clear      'Liste löschen
While lang <> 0          'Bis nichts mehr gelesen wird
    lang = GetPrivateProfileString("Termine", Trim$(Str$(i)), "",
buffer$, _ 255, _
    "termine.ini")
    If lang > 0 Then
        Termine.Liste.AddItem Left$(buffer$, lang) 'Termin hinzufügen
    End If
    i = i + 1
Wend
End Sub
```

Dies ist die andere Prozedur, welche die Datei wieder ausliest und die Terminliste füllt. Es wird so lange gelesen, wie die Länge der gelesenen Zeichen pro *Identifizier* ungleich Null ist.

Termineinträge aus Datei lesen

Eine weitere Prozedur, *Alte_Einträge_löschen*, kann aufgerufen werden, um Termine aus der Liste zu entfernen, die bereits abgelaufen sind. Dies bietet sich beim Start des Programms in der Ereignisprozedur *Form_Load* an. Hierin können Sie folgende Aufrufe platzieren:

```
Liste_füllen           'Gespeichertes in Liste laden
Alte_Einträge_löschen 'Abgelaufene Termine aus Liste löschen
Schreibe_neue_Liste   'Geänderte Liste wieder speichern
Liste_füllen          'erneut Laden, ohne abgelaufene Termine
```

Ebenfalls in der *Form_Load*- oder in der *Activate*-Ereignisprozedur können Sie die Textfelder für Datum und Uhrzeit mit den aktuellen Werten füllen.

Zum Schluss werden noch Verschönerungen und Verbesserungen eingebaut, die dem Benutzer mehr Komfort verschaffen. Mit dem Setzen der *ToolTip*-Eigenschaft werden einige Objekte beschrieben.

ToolTip

6.18.3 Übung: Stoppuhr – Zeiten stoppen und monetär bewerten



Um u.a. Arbeitsprozesse, Maschinenlaufzeiten, Schleifendurchläufe innerhalb eines Computerprogramms, Synchronisationsvorgänge, die Dauer eines Telefongesprächs etc. messen bzw. stoppen zu können, wird in der Regel eine Stoppuhr oder eine Uhr mit Zeitmessfunktionalität benötigt.

Damit Sie aber nicht mit einer Uhr neben Ihrem PC sitzen müssen, soll eine elektronische Stoppuhr entwickelt werden, mit der Sie diese Vorgänge professionell ausführen können.



Entwicklung eines Moduls, das in Ihre Anwendungen eingebunden und über die Methode *Show* aufgerufen werden kann.

Dieses Tool *Stoppuhr* soll außerdem so konzipiert werden, dass es für sich alleine schon autonom lauffähig ist, aber genauso gut auch in andere beliebige Applikationen problemlos eingebunden und durch nur einen Aufruf benutzt bzw. aktiviert werden kann.

Da es in der täglichen privaten wie auch beruflichen Praxis außer dem Stoppen von Zeiten, also dem reinen Zeitmessen, sehr oft erwünscht ist, die gemessene Zeit bzw. die vergehende Zeit monetär zu bewerten, soll auch dieses Feature als zusätzliches Ihren Anforderungen entsprechend erweiterbares Modul implementiert werden (Abbildung 6.110).

Abbildung 6.110:
Stoppen und
Bewerten



Somit wären wir dann mit diesem Modul in der Lage, beliebige Prozesse (zum Beispiel wie schnell läuft ein Programm ab, wie schnell wird das Bild am Monitor aufgebaut usw.) in Bezug auf die Dauer zu messen und diese Dauer in »Geld« ausgedrückt zu bewerten (Abbildung 6.110).



Stellen Sie sich dazu vor, Sie sitzen an Ihrem PC (geschäftlich oder privat) und arbeiten an einem wichtigen Arbeitsprozess. Dabei werden Sie von einem Kollegen oder Kunden etc. angerufen.

Mit diesem Modul ist Ihnen die Möglichkeit gegeben, die Dauer dieses Telefongesprächs, wenn gewünscht, festzuhalten und die Zeitdauer dieser »Störung« betriebswirtschaftlich (z.B. auf Basis Ihres Stundenlohnes, Tagessatzes etc.) zu bewerten (Abbildung 6.110).

Anders ausgedrückt erhalten Sie zwangsläufig von diesem Modul eine innerbetriebliche Rechnung über die durch diese Störung (immer vorausgesetzt, dass Ihnen ein Weiterarbeiten an Ihrem eigentlichen Arbeitsprozess dadurch nicht mehr möglich war) entstandenen Kosten.



Oder stellen Sie sich vor, Sie sitzen am PC (geschäftlich oder privat) und haben einen Kollegen oder Kunden etc. anzurufen. Mit diesem Modul ist es Ihnen dann ein Leichtes und immer visuell vor »Augen«, die vergangene Zeit und die dadurch aufgelaufenen Telefonkosten abzulesen (Abbildung 6.110).

- ▶ Professionelles und komfortables Messen von Zeitabschnitten.
- ▶ Die ablaufende Zeit soll zusätzlich monetär bewertet werden.
- ▶ Der Zeitmessvorgang (Starten und Stoppen eines Zeitabschnittes) soll unabhängig von der aktuellen PC-Systemzeit geschehen, das heißt, dass der Lauf der Systemzeit in keiner Weise beeinträchtigt oder verändert wird.
- ▶ Außerdem soll der Zeitmessvorgang quasi parallel laufen, sprich asynchron zu allen anderen Windows-Tasks. Die Multitaskingfähigkeit soll ausgenutzt werden, so dass, während quasi gleichzeitig andere Windows-Applikationen arbeiten, die Stoppuhr im Hintergrund »weitertickt«.
- ▶ In diesem Toolmodul soll innerhalb einer globalen Funktion eine 8-Kanal-Stoppuhr implementiert werden, so dass damit acht voneinander unabhängige Stoppuhren zur Verfügung stehen.
- ▶ Somit lässt sich diese Funktion unter anderem auch sehr gut zur Synchronisation, damit bestimmte Bewegungen auf verschiedenen Rechnern gleich schnell ablaufen, nutzen.
- ▶ Die Auflösung bzw. die kleinste mögliche messbare Zeit soll 55 ms betragen.
- ▶ Die maximal messbare Zeit soll, abhängig vom Systemstart, ca. 24 Tage und 20,5 Stunden betragen.
- ▶ Intuitive Bedienoberfläche.
- ▶ Über einen Windows-Editor, z. B. Notepad, sollen in einer externen Textdatei wichtige Startparameter für die Benutzeroberfläche und den internen Rechenvorgang hinterlegt werden können.
- ▶ So soll in dieser Datei der zu messende Arbeitsprozess textuell bestimmt werden können. Dieser Arbeitsprozess, zum Beispiel »Dauer eines Telefongespräches messen und bewerten«, soll Ihnen innerhalb der Benutzeroberfläche als Formüberschrift automatisch generiert werden.
- ▶ Zusätzlich sollen in dieser Bewertungsdatei die Währung z. B. DM, sFr etc. sowie ein monetärer Wert für die interne Verrechnung und Bewertung der vergangenen Zeit hinterlegt werden können.
- ▶ In einem eigens dafür vorgesehenen abgeschlossenen Bewertungs-Modul (BAS-Modul) sollen Sie Ihren Anforderungen entsprechend für den in der Bewertungsdatei hinterlegten monetären Wert ein Zeitraster zur Berechnung der Kosten programmieren können (ein Muster ist anbei). In andere Programmteile soll nicht eingegriffen werden müssen.
- ▶ Damit haben Sie mit diesem Tool die verbrauchende Zeit und die dadurch entstehenden Kosten immer im Blickwinkel Ihres Handelns.

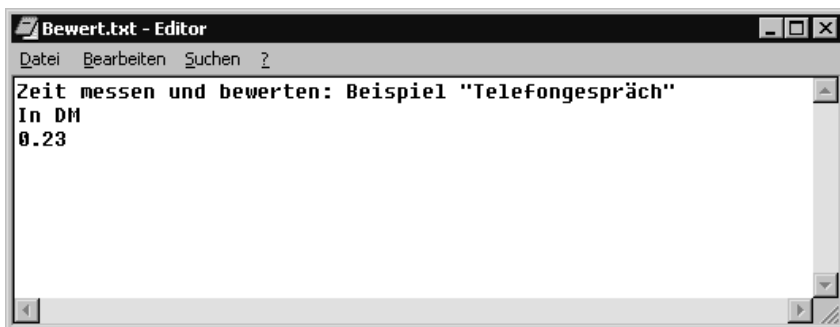
- ▶ Durch den modularen und übersichtlichen Programmaufbau lässt sich dann dieses Tool bequem in andere Applikationen integrieren und so für beliebige weitere praktische Anwendungsfälle aus dem technischen, betriebswirtschaftlichen, sportlichen oder rein privaten Umfeld nutzen.

Lösung

Damit das Stoppmodul seine Arbeit aufnehmen kann, erstellen wir mit einem beliebigen Editor (z.B. Notepad), der im ASCII/ANSI-Format speichert, folgende Datei:

Bewertungsdatei BEWERT.TXT

Abbildung 6.111:
Mit dem Notepad
können Sie Ihre
individuelle Bewer-
tungsdatei erzeu-
gen, die als
Grundlage vom
Stoppmodul zur
Anzeige und zur
internen Berech-
nung heran-
gezogen wird



Das Tool liest diese Bewertungsdatei (Abbildung 6.111) und erhält Informationen darüber, welche Überschrift (Arbeitsprozess) und Währung anzuzeigen und welcher monetärer Wert zur Bewertung herangezogen werden soll. Damit ist eine variable Grundsteuerung des Stoppmoduls von außen gewährleistet.

Konvention

Ab Zeile 1, Spalte 1 beginnend
Zeile 1 Bewertungstext - Benennung des zu stoppenden Arbeitsprozesses
Zeile 2 Benennung der Währung
Zeile 3 Monetärer Wert, der zur internen Berechnung innerhalb eines programmtechnisch zu definierenden Intervalls herangezogen wird

Einbinden

Das STOPPEN-Projekt enthält die Module *STOPPEN.FRM* zur Anzeige, *BEWERTEN.BAS* mit der Verrechnung und Bewertung sowie *STOPPUHR.BAS*, das den Algorithmus der 8-Kanal-Stoppuhr und deren Verarbeitung enthält.

Benötigen Sie für Ihre eigenen Projekte eines dieser Module, so binden Sie dieses ein, indem Sie Visual Basic mit Ihrer Applikation laden und danach über den Menü-Befehl **MODUL HINZUFÜGEN** das jeweilige Modul in Ihr Programmsystem integrieren.



Das Projekt STOPPEN ist aber auch so konzipiert worden, dass es für sich allein (autonom) schon lauffähig ist, es also nicht zwingend notwendig wird, die einzelnen Module in ein anderes VB-Projekt einzubinden.

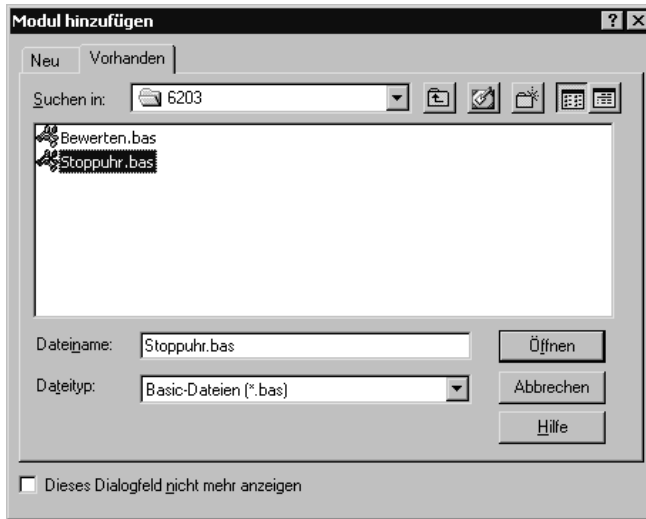
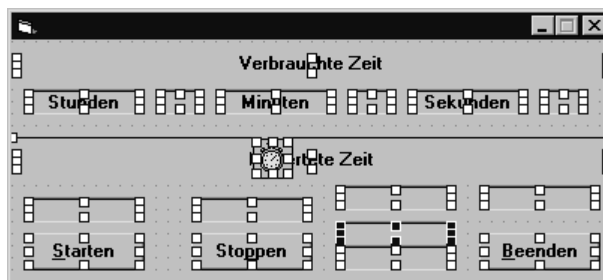


Abbildung 6.112:
BAS-Modul
hinzufügen

In das Projekt TEST, das eine Form TEST.FRM enthält, soll das Modul STOPP-UHR.BAS miteingebunden werden :



1. Visual Basic mit TEST.FRM laden.
2. Menü-Befehl MODUL HINZUFÜGEN aus dem Menü PROJEKT auswählen.
3. Modul STOPPUHR.BAS einladen (Abbildung 6.112).
4. Programmsystem TEST.FRM bei Beendigung speichern (Projektdatei TEST.VBP wird für Ihre Applikation neu erstellt oder modifiziert). Somit kann bei erneutem Aufruf von VB TEST das Modul STOPPUHR.BAS automatisch mitgeladen werden.



**Formular
Stoppen**

Abbildung 6.113:
Das Formular
»Stoppen«

Wie in der Abbildung 6.113 zu sehen ist, sind primär die Steuerelemente Bezeichnungsfeld (*Label*) im Hauptformular integriert. Danach stellt man sich bestimmt die Frage, ob die Funktion *Stoppuhr* mit dem Steuerelement *Timer* realisiert werden kann?

Der Timer ist dazu da, bestimmte Aktionen nach Ablauf einer definierten Zeitspanne, z.B. nach einer Sekunde oder nach einer Minute, automatisch zu starten, also folglich für die Funktion einer *Stoppuhr* absolut unbrauchbar, da hier eine Zeitspanne gemessen werden soll.

Im Formular ist das Steuerelement *Timer* nur deshalb eingebunden (Abbildung 6.113), um die aktuelle Uhrzeit und das aktuelle Datum im Sekundentakt einzublenden.

Die Funktion einer *Stoppuhr* kann also nicht nur mit den Mitteln von Visual Basic gelöst werden. Dazu wird eine speziell für diese Zwecke in Windows integrierte Funktion mit dem Namen *GetTickCount* aus der Bibliothek *Kernel32* in die *Visual Basic-Applikation* eingebunden und benutzt.

Modul STOPPUHR.BAS

Wenn Sie die Stoppuhr auch in einem anderen Projekt benutzen wollen, achten Sie bitte darauf, dass sich die nachfolgende Declare-Anweisung stets in einer Zeile im Deklarationsabschnitt einer Form oder eines Moduls befindet, bevor die Prozedur aufgerufen wird.

```
Public Declare Function GetTickCount Lib "kernel32" () As Long
```

Die Funktion »Stoppuhr« beinhaltet eine 8-Kanal-Stoppuhr, d.h. es stehen Ihnen acht voneinander unabhängige Stoppuhren zur Verfügung, die mit den folgenden Parametern angesteuert werden:

- ▶ KANAL vom Typ Integer: Bestimmt, welcher Kanal benutzt werden soll. Zulässige Werte von 1 bis 8.
- ▶ START vom Typ Integer: *True* steht für Uhr starten, *False* steht für Uhr auslesen.

Wenn Sie die Uhr starten, so ist der Funktionswert = 0. Beim Auslesen der Stoppuhr enthält der Funktionswert die abgelaufene Zeit in Sekunden, die Zeit läuft aber intern weiter, wodurch auch ein weiteres Zeit-Auslesen, z.B. für Zwischenzeiten etc., möglich ist.

```
Static Function StoppUhr (N As Integer, Start As Integer) As Single
Dim Begin(1 To 8) As Long
    If Start Then
        Begin(N) = GetTickCount()
        StoppUhr = 0
    Else
        StoppUhr = (GetTickCount() - Begin(N)) / 1000
    End If
End Function
```

Sollte die Stoppuhr-Funktion ausgelesen werden, ohne dass diese zuvor einmal gestartet worden ist, enthält der Funktionswert die Zeit ab dem Start von Windows.

In der folgenden Prozedur werden die Parameter für die Stoppuhr-Funktion versorgt und die Funktion im Multitasking (*DoEvents*) ständig aufgerufen. Der

übergebene Funktionswert ist die vergangene Zeit in Sekunden, welche noch zusätzlich in Minuten und Stunden umgerechnet wird.

Diese Zeitinformationen werden in die entsprechenden Steuerelemente *Label* zur Bildschirmausgabe gestellt und die Prozedur *Zeit_Bewerten* aus dem BAS-Modul *Bewerten.Bas* aufgerufen.

```

Sub Zeit_Auslesen ()
  Start% = False
  Kanal% = 1
  DoEvents
  Nochmal:
  If Merker = 2 Then
    Exit Sub
  End If
  If Merker = 1 Then
    Stoppen.Stopz.Caption = Time$
    Stoppen.Stopz.Visible = True
    Exit Sub
  End If
  DoEvents
  Zeit! = StoppUhr(Kanal%, Start%)
  Se = Int(Zeit!)          ' Gesamtzeit in Sekunden
  Stoppen.Stunden.Caption = Int(Se / 3600) ' Gesamtzeit in Stunden
  Stoppen.Minuten.Caption = Int(Se / 60)  ' Gesamtzeit in Minuten
  Stoppen.Sekunden.Caption = Se Mod 60 ' Den Rest in Sekunden
  Call Zeit_Bewerten(Ge!)
  Stoppen.Gesamt.Caption = Ge!
  GoTo Nochmal
End Sub

```

In dieser Prozedur programmieren Sie, entsprechend Ihrer Anforderung, wie und mit welchem Algorithmus das Zeitraster oder die vergangene Zeit monetär bewertet werden soll. Auch diese Prozedur wird ständig aufgerufen, so dass Sie immer die durch den Zeitverbrauch entstehenden Kosten am Bildschirm sehen.

**Modul
BEWERTEN.BAS**

Hier in diesem Beispiel berechnet die Prozedur basierend auf dem in der Datei *BEWERT.TXT* hinterlegten *WERT* (0,23 DM) die Gesamtkosten für jede angefangene Minute.

Beispiel

```

Sub Zeit_Bewerten (Ge As Single)
  Ge = 0
  W = Val(DS$(3)) ' Wert pro angefangene Minute
  MN = Int(Se / 60) ' Gesamtzeit in Minuten
  S = Se Mod 60 ' Den Rest in Sekunden
  If S <> 0 Then
    Ge = (MN + 1) * W
  Else

```

```

        Ge = MN * W
    End If
End Sub

```

Start Bevor Sie die Stopp-Applikation erfolgreich starten und benutzen können, müssen Sie in der Steuerdatei *BEWERT.TXT* die für Sie relevanten Einträge, Ihren gewünschten Anforderungen entsprechend, setzen.

Abbildung 6.114:
STOPP-Modul ist aufgerufen, aber ein Stoppvorgang noch nicht aktiviert worden



Rufen Sie *STOPPEN.EXE* auf, so werden Ihnen die aktuelle Systemzeit und das aktuelle Systemdatum angezeigt. Möchten Sie Ihren zu stoppenden Arbeitsprozess, hier in diesem Beispiel Telefongespräch, starten, so betätigen Sie die Schaltfläche *STARTEN* (Abbildung 6.114).

Die ablaufenden, verbrauchenden Sekunden, Minuten und Stunden sowie die monetäre Bewertung dessen werden Ihnen ständig am Bildschirm angezeigt und aktualisiert, so dass Sie Ihren Arbeitsprozess in Bezug auf Zeit und Kosten immer voll im Blick haben.

Abbildung 6.115:
Haben Sie Ihren Arbeitsprozess beendet, d.h. den Vorgang abgestoppt, so ersehen Sie die verbrauchte Zeit und die dafür entstandenen Kosten



Das Stoppmodul ist so entwickelt worden, dass jederzeit in andere Windows-Anwendungen gewechselt werden kann, ohne dass der Stoppvorgang unter-/abgebrochen oder die interne Systemzeit verändert wird. Die Zeit und die interne Bewertung laufen quasi parallel weiter.

Haben Sie Ihr Telefongespräch oder allgemein Ihren Arbeitsprozess beendet, klicken Sie auf die Schaltfläche *STOPPEN*, worauf Ihnen die verbrauchte Gesamtzeit und die entstandenen Gesamtkosten eingeblendet werden (Abbildung 6.115)

Zusätzlich sehen Sie über der Schaltfläche STARTEN, wann Sie Ihren Stoppvorgang begonnen, und über der Schaltfläche STOPPEN, wann Sie Ihren Stoppvorgang beendet haben.

6.19 Das Steuerelement »Image«

Das Anzeige-Element (Image) ist für die Anzeige von Bildern gedacht. Die Dateiformate entsprechen denen vom Steuerelement *Bildfeld*. Sein Aussehen wird vom jeweiligen Bild bestimmt. Es selbst kann, im Gegensatz zum Steuerelement *Bildfeld*, die Größe des Bildes beeinflussen.

6.19.1 Übung: Auf Anforderung ein Bild laden, anzeigen und dessen Größe anpassen



Abbildung 6.116:
Geladenes Bild
Plus!.Bmp

Über eine Befehlsschaltfläche (*CommandButton*) soll die Bitmap-Datei *Plus!.Bmp* aus dem Ordner *Windows* der Festplatte *C:* in das Anzeige-Element geladen werden. Dabei ist das zu ladende Bild automatisch auf die Größe des Steuerelements *Image* im Formular anzupassen (Abbildung 6.116).

Lösung

Damit bei Mausklick auf den Button auch das gewünschte Bild im Anzeige-Steuerelement erscheint, ist die Ereignisprozedur *Click* des *CommandButtons* zu füttern.

```
Private Sub Command1_Click()  
    Image1.Picture = LoadPicture("C:\windows\Plus!.bmp")  
End Sub
```

Der Eigenschaft *Picture* des Images ist der genaue Standort (*C:\windows\Plus!.bmp*) des anzuzeigenden Bildes zu übergeben. Die Funktion *LoadPicture* veranlasst das entsprechende Laden.

Das zu ladende Bild *Plus!.bmp* ist größer als der bildaufnehmende Rahmen, so dass nur ein Ausschnitt des Bildes im Anzeige-Steuerelement zu sehen wäre.

Da aber das Bild vollständig im Image erscheinen soll, muss das zu ladende Bild automatisch auf die Größe des Steuerelements *Image* im Formular angepasst werden. Dazu ist die Ereignisprozedur *Click* des *CommandButtons* noch wie folgt zu ändern:

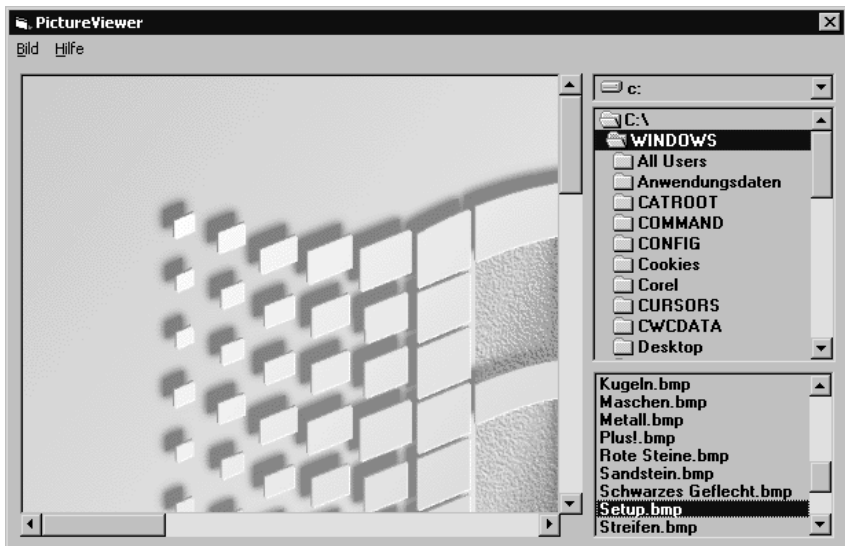
```
Private Sub Command1_Click()  
    Image1.Stretch = True  
    Image1.Picture = LoadPicture("C:\windows\Plus!.bmp")  
End Sub
```



Mit der Eigenschaft *Stretch* legen Sie fest, ob die zu ladende Grafik entsprechend der Größe des Anzeige-Steuerelements (*Stretch = True*) oder das Anzeige-Steuerelement entsprechend der Größe der Grafik (*Stretch = False*) zu skalieren ist.

Starten Sie nun Ihre Applikation z.B. über **[F5]** und aktivieren den Button *Bild laden und Größe anpassen*, so erscheint, wie in Abbildung 6.116 zu sehen, das Plus!-Bitmap vollständig im Anzeige-Steuerelement.

Abbildung 6.117:
Der Bildbetrachter
(PictureViewer)



6.19.2 Übung: Ein Bildbetrachter (PictureViewer) gefällig

Vermutlich haben Sie auch das Problem, dass sich bei Ihnen immer mehr Dateien auf der Festplatte ansammeln und Sie langsam den Überblick darüber verlieren, was Sie davon noch gebrauchen können.

Ein nicht unwesentlicher Teil dieser Dateien sind Bilddateien. Manche Programme integrieren Bilder in die .exe-Datei, andere liefern die Bilder als einzelne Dateien mit.

Diese Bilder sind in der Regel Symbole, Hintergrundbilder und weitere Bilddateien, die zur Verschönerung der Applikationen und von Windows selbst dienen. Für jemanden, der sich gerne mit Grafiken beschäftigt, gibt es inzwischen eine schier unendliche Auswahl an Bildern auf CDs und dies vor allem im Internet.

Das wird zum Problem, wenn Sie sich diese Bilder anschauen wollen. Schnell stellt sich die Frage, womit dies geschehen soll. Das von Microsoft zu Windows mitgelieferte Paintbrush ist sicher nicht das richtige Programm.

Und jedes Mal eine Grafiksoftware starten oder den Öffnen-Dialog durchklicken ist auch nicht gerade angenehm.

Daher wollen wir uns selbst helfen, indem wir einen reinen Bildbetrachter (PictureViewer) entwickeln (Abbildung 6.117). Hierfür legen wir uns folgende Leistungsmerkmale und Restriktionen fest:

- ▶ Sie sollen freien Zugriff auf die Dateistruktur aller Laufwerke haben.
- ▶ In einer Dateilistbox werden alle anzeigbaren Bilddateien aufgelistet.
- ▶ Es sollen Bilder vom Format *.BMP, *.ICO, *.WMF, *.GIF und *.JPG angezeigt werden können.
- ▶ Ein Vollbild-Modus ermöglicht das Anzeigen von großen Bildern.
- ▶ Sie können wählen, ob Sie das Bild an das Fenster anpassen wollen.
- ▶ Sie können Informationen über das Programm abrufen.
- ▶ Diese Optionen sollen per Menü einstellbar sein.

Leistungsmerkmale und Restriktionen

Nachdem nun diese Spezifikationen feststehen, überlegen wir, wie man dies objektorientiert in die Realität umsetzen kann. Zum Anzeigen wird also ein Objekt benötigt, das in der Lage ist, Bilder anzuzeigen und die angezeigte Größe zu verändern.

Müssten wir uns dieses Objekt selbst erzeugen, so hätte es sicher unter anderem die Eigenschaften *Bilddatei* und *Anzeigemodus*.

Wenn wir uns nun die mitgelieferten Steuerelemente ansehen, werden wir auf das *Anzeige*-Steuerelement (*Image*) stoßen. Es kann Bilddateien anzeigen und deren dargestellte Größe über die Eigenschaft *Stretch* anpassen (Abbildung 6.120).

Image oder PictureBox

Es hat allerdings den Nachteil, dass es bei abgeschaltetem *Stretch* die Bilder in Originalgröße darstellt, was manchmal zu groß wird. Das *Bildfeld* (*Picture*) hingegen zeigt immer nur den Ausschnitt eines Bildes, der in das Bildfeld passt.

Weitere benötigte Objekte wären ein Menüobjekt und ein Objekt zur Auswahl einer Datei im Dateisystem. Für diese Objekte existieren bereits Steuerelemente in Visual Basic. Damit wir die Steuerelemente verwenden und Objekte erzeugen können, benötigen wir ein Formular.

Anforderungsanalyse

Ein Formular soll unser Hauptfenster sein; daher sollte hier das Menü platziert werden. Als obersten Menü-Befehl können ein *Bild-* und ein *Hilfe-*Befehl vergeben werden.

Abbildung 6.118:
Das Menü *Bild*



Hilfe ist in jedem Programm vorhanden und sei es wie hier, dass man nur Informationen über das Programm erhält. Daher enthält der Menü-Befehl *Hilfe* nur den Unterbefehl *Info...* (Abbildung 6.119). UnterMenü-Befehle von Menü *Bild* (Abbildung 6.118) wären dann *Vollbild*, *Bild anpassen* und ein *Beenden*-Befehl.

Abbildung 6.119:
Über Menü-Befehl
Info



Damit die Programminformationen angezeigt werden können, brauchen wir ein weiteres Formular (Abbildung 6.119). Ein drittes Formular wird möglicherweise notwendig, wenn wir uns überlegen, wie der Vollbild-Modus realisiert werden soll.

Dabei wird ein Objekt benötigt, das in der Lage ist, den gesamten Bildschirm auszufüllen. Prinzipiell ist hierzu das Steuerelement *Image* in der Lage. Man müsste das Formular auf Vollbild umstellen und das Steuerelement *Image* dann in das Formular einpassen.

Ein Formular kann jedoch auch Bilddateien anzeigen, was jedoch zu Folge hätte, dass man ein eigenes Formular für den Vollbild-Modus benutzen oder das bestehende Formular ändern müsste. Durch solche Überlegungen prüfen Sie Ihre Analyse gegen die Realisierbarkeit und können sich für die optimale Methode entscheiden.

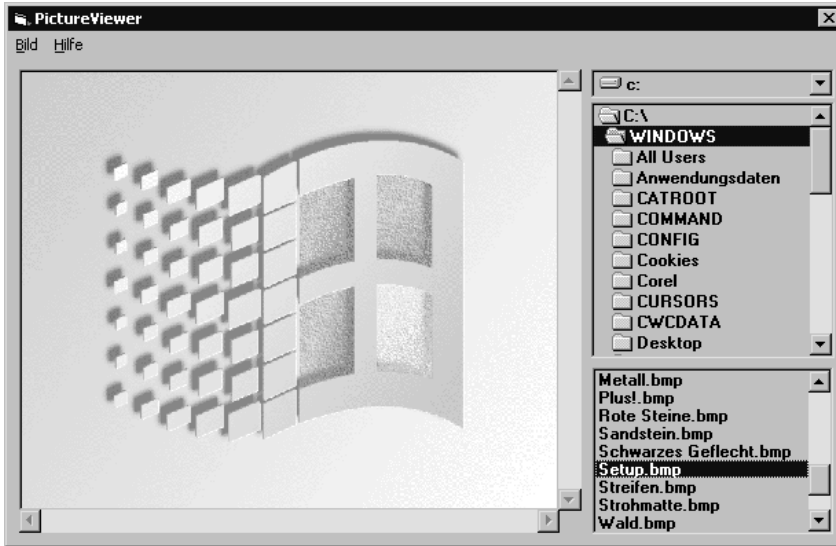


Abbildung 6.120:
Bild der Image-
größe anpassen

Wir entscheiden uns für den Weg der Vollbildanzeige im gleichen Formular, mit einem *Image*-Steuerelement und einem *Bildfeld*-Steuerelement. Dies wird notwendig, weil wir die Eigenschaft *Stretch* des Image-Feldes nutzen (Abbildung 6.120), aber andererseits die Größe des Bildes beschränken wollen.

Image und PictureBox

Lösung

Legen wir also zuerst das Projekt an und wählen dabei ein Standard-EXE-Projekt aus. Das Formular wird automatisch angezeigt. Betrachten wir dieses Formular als unser Hauptfenster.

Sie sollten zunächst das Menü hinzufügen. Klicken Sie mit der rechten Maustaste in das Formular und öffnen Sie im Pop-Up-Menü den Befehl *Menü-Editor...* Der Menü-Editor lässt sich nur starten, wenn ein Formular bearbeitet wird. Vergeben Sie nun einen Menü-Befehl mit der Caption *Bild* und mit dem Namen *Men_Bild* (Abbildung 6.121).

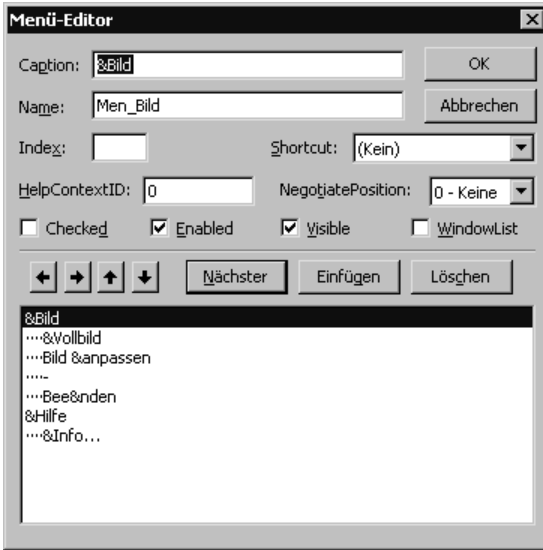
Fügen Sie mit dem Befehlsknopf *Nächster* einen neuen Menü-Befehl ein und schieben diesen mit der Formatierpfeiltaste *rechts* eine Einrückungsebene nach rechts. Dieser Menü-Befehl wird dadurch später unter dem *Bild*-Befehl erscheinen (Abbildung 6.121).

Nennen Sie diesen *Vollbild* und vergeben Sie den Namen *Men_Bild_Vollbild*. Durch eine solche Namensvergabe ist eine Zuordnung des Namens zum Steuerelement einfacher.

Fügen Sie wieder mit *Nächster* einen weiteren Befehl ein. Die Einrückungsebene bleibt hierbei erhalten. Nennen Sie ihn *Bild anpassen* und vergeben Sie den Namen *Men_Bild_anpassen*. Fügen Sie einen weiteren Befehl ein. Dies soll ein Trennstrich werden (Abbildung 6.121).

Menü-Befehle

Abbildung 6.121:
Menüdefinitionen
mit dem Menü-
Editor



Ein solcher besteht in der Caption nur aus einem Bindestrich. Allerdings müssen Sie auch hierfür einen Namen vergeben, also *Men_Bild_Strich*. Darunter fügen Sie den *Beenden*-Befehl mit *Men_Bild_Beenden* als Namen ein.

Der nächste Menü-Befehl, den Sie einfügen, muss mit der Formatierpfeiltaste auf die Ebene von *Bild* gebracht werden. Dies wird die Menüüberschrift *Hilfe* mit dem Namen *Men_Hilfe*. Darunter schließlich fügen Sie eine Ebene eingerückt den Befehl *Info...* mit dem Namen *Men_Hilfe_Info* ein (Abbildung 6.121).

Beenden Sie den Menü-Editor mit der Schaltfläche OK. Sie sehen, dass das Menü sofort in das Formular eingebaut wird (Abbildung 6.122).

weitere Steuer- elemente

Nun fügen Sie die weiteren Steuerelemente mit Hilfe der Toolbox ein. Dies wären ein *Image-Control*, ein *Bildfeld*, eine *Laufwerkslistbox*, eine *Verzeichnislistbox* und eine *Dateilistbox*. Machen Sie das Formular so groß, wie Sie wollen und entsprechend dem zur Verfügung stehenden Platz (Abbildung 6.122).

Platzieren Sie die drei Listboxen an der rechten Seite Ihres Formulars untereinander (Abbildung 6.122). Optimal liegt die *Laufwerkslistbox* oben, dann die *Verzeichnislistbox* und die *Dateilistbox* unten.

Platzieren Sie das Bildfeld in der linken oberen Ecke, mit geringem Abstand, und ziehen Sie es in der Größe bis fast an die Listenfelder und den unteren Rand des Formulars.

PictureBox und Image

Wählen Sie nun das Steuerelement *Image* an und entfernen Sie es mit dem Menü-Befehl BEARBEITEN->AUSSCHNEIDEN oder den Tasten `[Umschalt]` `[Entf]`.

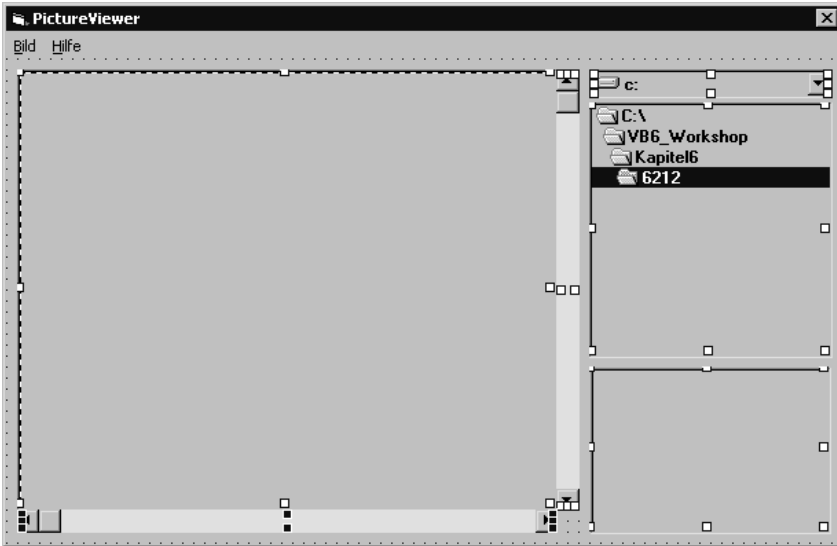


Abbildung 6.122:
Das Formular mit
seinen Steuer-
elementen

Wählen Sie nun das Bildfeld an, und fügen Sie das Image-Feld wieder ein. Das Image-Feld wird in der linken oberen Ecke eingefügt, welche die neuen Ursprungskordinaten 0,0 für das Image-Feld sind.

Geben Sie dem Image-Feld die *Height*- und *Width*-Eigenschaften gleich den Werten des Bildfelds. Nun ist das Image-Feld exakt in das Bildfeld eingefügt. Die Eigenschaft *BorderStyle* beim Steuerelement *Image* muss auf 0 stehen, hat also keinen Rahmen. Den Rahmen soll das Bildfeld liefern.

**das Formular
»Programm-
informationen«**

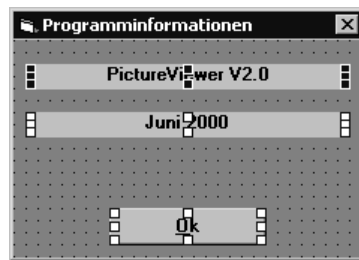


Abbildung 6.123:
Das Formular
»Programm-
informationen«

Erzeugen Sie nun ein weiteres Formular, in das die Programminformation geschrieben werden soll (Abbildung 6.123). Die Befehlsschaltfläche entlädt im Ereignis *Click* dieses Formular wieder.

Geladen wird es im Menü-Befehl *HILFE->INFO...* mit der Methode *Show*. Im Ereignis *Load* des Formulars kann man dieses noch so positionieren, dass es mittig über dem *PictureViewer*-Formular erscheint.

Bilddateihandling Das Bild einer Bilddatei soll angezeigt werden, wenn eine entsprechende Datei in der Dateilistbox ausgewählt wird. Folglich muss die Prozedur zum Aufbau des Bildes in der Ereignisprozedur *Click* des Dateilistfeldes geschrieben werden.

Das Dateilistfeld benötigt aber für die Anzeige von Dateien bestimmte Eigenschaften. Dies ist der Pfad des Verzeichnisses, in dem nach möglichen Bilddateien gesucht werden soll.

Welcher Art die Dateien sind, sprich welche Erweiterung sie haben, kann über die Eigenschaft *Pattern* eingestellt werden. Hier lassen sich auch mehrere Dateierweiterungen miteinander angeben.

Bildformate Die Eigenschaft *Picture* der Steuerelemente, die Bilder anzeigen können, erlaubt seit der Version 5.0 von Visual Basic Bilder vom Typ **.BMP*, **.ICO*, **.WMF*, **.GIF* und **.JPG*. Diese Erweiterungen können in der Eigenschaft *Pattern* des Dateilistfeldes angegeben werden, wodurch Sie nur die Bilddateien angezeigt bekommen, die das Steuerelement *Image* auch anzeigen kann.

Das Dateilistfeld kann die benötigte Pfadinformation zum Verzeichnis vom Verzeichnislistfeld bekommen. Dies hat die Eigenschaft *Path*, welche direkt an das Dateilistfeld übergeben werden kann.

Das Verzeichnislistfeld verfügt nicht über ein Ereignis *DbClick*, obwohl die Verzeichnisse per Doppelklick geöffnet werden. Dafür hat es das Ereignis *Change*, das immer dann auftritt, wenn sich der Pfad geändert hat. Die Übergabe des Pfades an das Dateilistfeld kann also hier erfolgen.

Das Verzeichnislistfeld seinerseits benötigt als Vorgabe das zu verwendende Laufwerk. Diese Information kann das Laufwerklistfeld liefern. Die Eigenschaft *Drive* kann hierbei direkt an die Eigenschaft *Path* des Verzeichnislistfeldes übergeben werden.

Auch das Laufwerklistfeld verfügt über ein *Change*-Ereignis, das für diese Zwecke geeignet ist. Somit sind die drei Objekte zur Dateiauswahl wie folgt miteinander verknüpft:

```
Private Sub Laufwerke_Change()  
    Verzeichnisse.Path = Laufwerke.Drive 'Pfad für  
Verzeichnislistbox  
End Sub  
Private Sub Verzeichnisse_Change()  
    Dateien.Path = Verzeichnisse.Path 'Pfad weitergeben  
End Sub
```

Code zum Anzeigen des Bildes Nun können wir den Code zum Anzeigen des Bildes in die Ereignisprozedur *Click* von *Dateien* hinzufügen. Dieser ist sehr einfach, da die Eigenschaft *Picture* des Steuerelements *Image* solche Mechanismen übernimmt. Es genügt, dieser Eigenschaft eine Bilddatei zuzuweisen. Hierfür muss die Funktion *LoadPicture* von Visual Basic verwendet werden:

```
Image1.Picture = LoadPicture(Dateil.Path & "\" & Dateil.FileName)
```

In der Klammer wird aus der Eigenschaft *Path* und dem Dateinamen durch Verbinden mit einem »\« der Pfad zur Bilddatei.

Dies funktioniert nicht mit Bilddateien, die im Ursprungsverzeichnis eines Laufwerks liegen.



Den obigen Ausdruck kann die Funktion *LoadPicture* verwenden und einen Bezug auf die Bilddatei an das Image-Steuerelement und seine Eigenschaft *Picture* geben.

Diesen Teil des Programms können Sie bereits testen. Starten Sie einfach mit der Taste [F5]. Wählen Sie ein Verzeichnis, in dem sich Bilddateien befinden, und klicken Sie auf eine der angezeigten Dateien.

starten und probieren

Das Bild wird unmittelbar nach dem Klick im Image angezeigt. Je nach Größe des Bildes sehen Sie es vollständig oder nur den linken oberen Teil. Da dies bei großen Bildern unbefriedigend ist, gibt es zwei Möglichkeiten, das Bild doch komplett anzuzeigen. Dies wären der Vollbild-Modus und das Anpassen des Bildes.

Zunächst wollen wir das Bild per Menü-Befehl an die Größe des Bildfeldes anpassen. Hierzu bearbeiten wir die Ereignisprozedur *Click* des Menü-Befehls BILD ANPASSEN. Jeder Menü-Befehl hat die Eigenschaft *Checked*. Hierüber kann ein Häkchen (*Checked = True*) an den Menü-Befehl gesetzt werden, um einen Zustand optisch zu markieren.

Bildgröße anpassen

Wenn wir den Menü-Befehl anwählen, soll der Zustand wechseln. Daher muss dieser bei jedem Ereignis in das Gegenteil des aktuellen Zustands gebracht werden. In dessen Abhängigkeit schalten wir die Eigenschaft *Stretch* des Steuerelements *Image* ein bzw. aus. *Stretch=True* passt die Größe des Bildes automatisch an die Größe des *Image*-Objekts an.

strecken ja oder nein

Bei *Stretch=True* wird die letzte Größe des Steuerelements *Image* beibehalten und das neue Bild dort hineinprojiziert. Also sollte bei aktiviertem *Stretch* das Image wieder in die Ausgangsgröße gebracht werden.



Die zugehörige Ereignisprozedur sieht dann so aus:

```
Private Sub Men_Bild_anpassen_Click()
    If Bildmain.Men_Bild_anpassen.Checked = True Then 'bereits
        aktiviert?
        Image1.Stretch = False 'wenn ja, dann ausschalten
        Bildmain.Men_Bild_anpassen.Checked = False 'Häkchen entfernen
    Else
        Image1.Stretch = True 'Wenn nein, dann einschalten
        Bildmain.Men_Bild_anpassen.Checked = True 'Häkchen hinzufügen
        Image1.Width = Picture1.Width 'Bildfeld wieder anpassen
        Image1.Height = Picture1.Height 'Bildfeld wieder anpassen
        Image1.Top = 0 'Bildfeld positionieren
        Image1.Left = 0
    End If
End Sub
```

Vollbild-Anzeige Wem diese Darstellung nicht reicht, der kann die Vollbild-Anzeige wählen. Die Realisierung erfolgt ebenfalls über einen Menü-Befehl, *Men_Bild_Vollbild*. In der Ereignisprozedur *Click* muss nun Folgendes geschehen:

Wir wollen als Anzeigemedium das Formular benutzen, da es den Vollbild-Modus als Eigenschaft besitzt. Das bedeutet aber, dass alle Objekte im Formular unsichtbar gemacht werden müssen, da sie sonst im Bild erscheinen würden. Hierfür bietet sich eine kleine Prozedur an, die in einem Codemodul untergebracht ist.

```
Sub sichtbar(mode As Integer)
    Bildmain.Picture1.Visible = mode
    Bildmain.Verzeichnisse.Visible = mode
    Bildmain.Laufwerke.Visible = mode
    Bildmain.Dateien.Visible = mode
    Bildmain.Men_Bild.Visible = mode
    Bildmain.Men_Hilfe.Visible = mode
End Sub
```

In *mode* entscheiden Sie, ob Sie die Objekte sichtbar oder unsichtbar darstellen wollen. Führen Sie alle Objekte, auch die Menüüberschriften, nacheinander auf und setzen Sie deren Eigenschaft *Visible* auf *mode*.



Sie können nun die Objekte mit *sichtbar False* auf unsichtbar und mit *sichtbar True* wieder auf sichtbar setzen, und das mit einer einzigen Prozedur. Ein wesentlich eleganterer und objektorientierter Ansatz wäre folgender:

```
Sub sichtbar(mode As Integer)
    Dim element As Object
    For Each element In Bildmain.Controls
        element.Visible = mode
    Next
End Sub
```

Für jedes Steuerelement im Formular *Bildmain* würde die Eigenschaft *Visible* auf den Wert von *mode* gesetzt. Das ist aber leider auch der Grund, weshalb dieser Ansatz einen Fehler produziert.

Für Menüelemente besteht die Einschränkung, dass mindestens ein Element die Eigenschaft *Visible=True* besitzen muss. Dabei ist es unsinnigerweise möglich, ein ganzes Menü unsichtbar zu machen, obwohl die Unterbefehle sichtbar sind. Sonst wäre der obige Ansatz möglich gewesen.

Bild in Formular laden

Tragen Sie also den Aufruf zum *Unsichtbarmachen* in das Menüereignis ein. Ebenfalls dort muss das Bild in das Formular geladen und die Vollbild-Darstellung eingeschaltet werden.

Das Bild laden Sie auf dem gleichen Weg wie beim Steuerelement *Image* mit der Funktion *LoadPicture*. Die Pfadangaben sind auch in unsichtbarem Zustand abrufbar.

Dann setzen Sie die Eigenschaft *WindowState* des Formulars auf 2, was dem Vollbild entspricht. Dies können Sie einmal testen. Dabei wird Ihnen sicherlich auffallen, dass Sie den Vollbild-Modus nun nicht mehr beenden können.

Wir benötigen daher einen Weg zum Wiederherstellen des vorherigen Zustands. Zu diesem Zweck nehmen wir das Ereignis *Click* des Formulars.

Wenn Sie mit der Maus in das Vollbild klicken, soll es wieder zur ursprünglichen Größe zurückkehren. Also fügen Sie in die Ereignisprozedur *Click* folgenden Code ein:

```
Private Sub Form_Click()
    If Picture1.Visible = False Then 'Wenn das Bildfeld unsichtbar ist
        Me.WindowState = 0          'Wieder Normaldarstellung einschalten
        Me.Picture = LoadPicture() 'Bild entfernen
        sichtbar True              'Alle Objekte wieder sichtbar machen
    End If
End Sub
```

Me ist ein Spezialobjekt in Visual Basic, das immer Bezug auf das aktuelle Formular hat. Somit setzt *Me.WindowState = 0* den *WindowState* des aktuellen Formulars auf *Normal*. Da der Code normalerweise zum aktuellen Formular aufgerufen wird, kann das *Me*-Objekt recht häufig verwendet werden.

**das Spezialobjekt
Me**

Weiterhin müssen Sie das Bild wieder aus dem Formular entfernen. Dies erledigt die Zuweisung *Me.Picture = LoadPicture()*, also ohne Argumente. Schließlich machen Sie die Objekte wieder mit dem Aufruf *Sichtbar True* sichtbar.



**Pfad ist nicht
gleich Pfad**

Es gibt nun noch einige Abfragen, die Sie einbauen können, damit es im Fehlerfall nicht zu einem Programmabsturz kommt. Hierzu müssen Sie die möglichen Fehlerquellen lokalisieren. Ein möglicher Punkt ist das Dateisystem. Normalerweise liefert die Eigenschaft *Path* einen Pfad zu einem Verzeichnis ohne den Backslash (»\«) am Ende.

Ein Hinzufügen eines Dateinamens zu diesem Pfad würde eine unverständliche Pfadangabe ergeben und das Öffnen der Bilddatei würde scheitern. Andererseits hat der Pfad zu einer Datei, die auf dem Basis-Verzeichnis eines Laufwerks liegt (also etwa *C:*), bereits einen Backslash.

Diesem kann natürlich nicht automatisch ein Backslash hinzugefügt werden. Sie müssen also das Zeichen rechts außen des Pfads dahingehend darauf untersuchen, ob es einem Backslash entspricht, und gegebenenfalls diesen hinzufügen.

Ein anderer Fehler, der auftreten kann, ist der, einen Eintrag einer Listbox zu verwenden, obwohl keiner ausgewählt ist. Dies kann auf jeden Fall dann passieren, wenn im gewählten Verzeichnis gar keine Bilddateien vorhanden sind.

**mögliche Listbox
Fehler**

Doch wenn Sie Bilddateien angezeigt bekommen, heißt das noch nicht, dass automatisch eine davon selektiert wird. Dass kein Eintrag gewählt wurde, erkennt man daran, dass die farbliche Markierung fehlt.

Es ist nur ein gestrichelter Rahmen um den ersten Eintrag zu sehen. Beim Abfragen des Inhalts der selektierten Zeile wird ein leerer String zurückgegeben. Die Eigenschaft *ListIndex* hat in diesem Fall den Wert *-1*.

Dies müssen Sie abfragen, um einen Fehler verhindern zu können. In diesem Fall kann der Fehler dann auftreten, wenn Sie in den Vollbild-Modus schalten wollen, ohne eine Datei im Dateilistfeld selektiert zu haben.

ListIndex Es gibt zwei Möglichkeiten, diesen Fehler zu verhindern: Entweder Sie fragen die Eigenschaft *ListIndex* der Dateilistbox ab, wie oben beschrieben, oder aktivieren den Menü-Befehl erst dann, wenn eine Datei selektiert wurde.

Jedes Steuerelement, und dazu zählen Menü-Befehle auch, hat die Eigenschaft *Enabled*. Diese auf *False* gesetzt unterbindet jegliche Einflussnahme seitens des Benutzers.

Vom Code aus lässt sich das Objekt weiterhin ansprechen. Somit wäre das Ereignis *Click* des Menü-Befehls so lange deaktiviert, wie noch keine Datei im Dateilistfeld selektiert wurde. Eine Datei selektieren Sie mit einem Klick, folglich gehört ein solcher Code ins Ereignis *Click* des Dateilistfelds. Hier steht dann die Anweisung:

```
Men_Bild_Vollbild.Enabled = True
```

Damit der Menü-Befehl nun nicht dauerhaft *enabled* ist, muss dieser Zustand wieder zurückgesetzt werden. Hierbei muss überlegt werden, wann der ListIndex im Dateilistfeld wieder zu *-1* werden kann.

Das geschieht beim Wechsel in ein anderes Verzeichnis, also beim Neuaufbau des Inhalts. Daher bietet es sich an, den Menü-Befehl im Ereignis *Change* des Verzeichnislistfelds wieder auf *enabled = False* zu setzen. Hier nun die Prozedur *Click* des Dateilistenfeldes:

```
Private Sub Dateien_Click()  
Dim Pfad As String, Faktor As Single  
    If Right$(Verzeichnisse.Path, 1) = "\" Then  
'Prüfen, ob "\"Zeichen im Pfad vorhanden, wenn nein hinzufügen  
        Pfad$ = Verzeichnisse.Path & Dateien.filename  
    Else  
        Pfad$ = Verzeichnisse.Path & "\" & Dateien.filename  
    End If  
    Image1.Left = 0 'Bildfeld positionieren  
    Image1.Top = 0 'Bildfeld positionieren  
    Image1.Stretch = False 'Stretch ausschalten  
    Image1.Picture = LoadPicture(Pfad$) 'Bild laden  
    DoEvents 'Rechenzeit geben, um das Bild aufzubauen  
    If Men_Bild_anpassen.Checked = True Then 'Bild muss angepasst werden  
        Faktor = Image1.Width / Image1.Height 'Verhältnis Länge/Breite  
        Image1.Stretch = True 'Stretch einschalten  
        Image1.Width = Picture1.Width 'Breite anpassen  
        Image1.Height = Int(Image1.Width / Faktor) 'Höhe anpassen  
        If Image1.Height > Picture1.Height Then 'Wenn höher als breit
```



```

        Image1.Width = Int(Picture1.Width * Faktor) 'Breite anpassen
        Image1.Height = Picture1.Height           'Höhe anpassen
    End If
End If
Men_Bild_Vollbild.Enabled = True'Menü-Befehl aktivieren
End Sub

```

Prinzipiell verfügen Sie nun über ein funktionsfähiges Programm. Man könnte jedoch den Komfort noch etwas erhöhen. Ein Bild, das zu groß für die Darstellung in der normalen Anzeige ist, könnte per Schiebebalken (*Scrollbars*) im sichtbaren Ausschnitt verschoben werden.

Scrollbars integrieren

Hierbei wird der Umstand, dass das Steuerelement *Image* innerhalb eines Steuerelements *PictureBox* liegt, zum Vorteil. Es ist hierdurch möglich, das Steuerelement *Image* zu verschieben, ohne dass es für den Betrachter seine Position verändert. Da es innerhalb des Steuerelements *PictureBox* liegt, stellt dies die sichtbare Grenze des Steuerelements *Image* dar.

Visual Basic stellt die Schiebebalken als Steuerelemente zur Verfügung. Es gibt sie horizontal und vertikal. Platzieren Sie je einen Schiebebalken rechts und unterhalb des Steuerelements *PictureBox* und passen Sie die Größe und Position an. Die Funktionsweise der Schiebebalken ist recht einfach. Die Richtung ist bereits über das Steuerelement vorgegeben.

Über die Eigenschaften einzustellen sind die Werte für einen maximalen und minimalen Wert, den der Schieberegler an den Enden annimmt. Der aktuelle Wert ist über die Eigenschaft *Value* verfügbar.

Schieberegler einstellen

Value nimmt also Werte zwischen den Eigenschaften *Max* und *Min* an. In diesem speziellen Fall muss der Bereich für *Value* zwischen 0 und dem Wert liegen, um den das Bild größer als der sichtbare Bereich ist. Diese Differenz lässt sich einfach berechnen und der Eigenschaft *Max* zuweisen.

Ein Schieberegler verfügt über zwei weitere Eigenschaften, die den Schiebeprozess steuern. Dies sind die Eigenschaften *SmallChange* und die *LargeChange*. *SmallChange* bestimmt, um wie viel *Value* verändert wird, wenn auf eine der Pfeiltasten mit der Maus geklickt wird.

SmallChange und LargeChange

LargeChange verändert ebenfalls den Wert von *Value*, jedoch dann, wenn mit der Maus in den Schieberegler geklickt wird. Auch wenn die Maustaste gedrückt bleibt, wird weitergeschoben.

In jedem Fall wird das Ereignis *Change* zum Schieberegler ausgelöst. In diesem Ereignis kann dann die eigentliche Aktion des Verschiebens erfolgen. Dies wäre für den horizontalen Balken:

```
Image.Left = 0 - Hscroll.Value
```

Und für den vertikalen Balken:

```
Image.Top = 0 - Vscroll.Value
```

Berechnen Sie *SmallChange* so, dass etwa 20 Ereignisse und bei *LargeChange* acht Ereignisse nötig sind, um vom einen zum anderen Ende des Schiebebalkens zu schieben.



Die Zuweisung der *Max*-Werte sollte dort erfolgen, wo das Bild geladen wird und die Größe bereits durch das Steuerelement *Image* ermittelt werden kann. Sinnvoll ist auch eine Abfrage dahin, ob das Bild größer als der sichtbare Bereich ist.

Ist das nicht der Fall, so sollten Sie die Schiebebalken mit der Eigenschaft *Enabled=False* deaktivieren, andernfalls mit *.Enabled=True* aktivieren.

Diese Abfrage kann Fehler verhindern und erspart die Rechenzeit zur Berechnung der Differenz für *Value* und die Eigenschaften *SmallChange*, *LargeChange*. Beachten Sie, dass Sie die Schiebebalken im Vollbild-Modus ebenfalls unsichtbar machen und später wieder einblenden müssen. Hier nun die vollständige Prozedur *Click* des Dateilistfeldes:

```
Private Sub Dateien_Click()
Dim Pfad As String, Faktor As Single
    If Right$(Verzeichnisse.Path, 1) = "\" Then
'Prüfen, ob "\"Zeichen im Pfad vorhanden, wenn nein hinzufügen
        Pfad$ = Verzeichnisse.Path & Dateien.filename
    Else
        Pfad$ = Verzeichnisse.Path & "\" & Dateien.filename
    End If
    Image1.Left = 0 'Bildfeld positionieren
    Image1.Top = 0 'Bildfeld positionieren
    Image1.Stretch = False 'Stretch ausschalten
    Image1.Picture = LoadPicture(Pfad$) 'Bild laden
    DoEvents 'Rechenzeit geben, um das Bild aufzubauen
    If Men_Bild_anpassen.Checked = True Then 'Bild muss angepasst werden
        Faktor = Image1.Width / Image1.Height 'Verhältnis Länge/Breite
        Image1.Stretch = True 'Stretch einschalten
        Image1.Width = Picture1.Width 'Breite anpassen
        Image1.Height = Int(Image1.Width / Faktor) 'Höhe anpassen
        If Image1.Height > Picture1.Height Then 'Wenn höher als breit
            Image1.Width = Int(Picture1.Width * Faktor) 'Breite anpassen
            Image1.Height = Picture1.Height 'Höhe anpassen
        End If
    End If
    Men_Bild_Vollbild.Enabled = True 'Menü-Befehl aktivieren
    DoEvents 'Rechenzeit, um Bild aufzubauen
    If Image1.Width > Picture1.Width Then 'Bild breiter als Bereich
        HScroll11.Enabled = True 'Horiz. Schiebepalken
        HScroll11.Max = Image1.Width - Picture1.Width + 90 'Max-Wert
festlegen
        HScroll11.Min = 0
        HScroll11.LargeChange = Int(HScroll11.Max / 3)
        HScroll11.SmallChange = Int(HScroll11.Max / 10)
    Else
```

```

HScroll1.Enabled = False 'sonst Schiebepalken deaktivieren
End If
If Image1.Height > Picture1.Height Then 'Vertikaler Balken
    VScroll1.Enabled = True
    VScroll1.Max = Image1.Height - Picture1.Height + 90
    VScroll1.Min = 0
    VScroll1.LargeChange = Int(VScroll1.Max / 3)
    VScroll1.SmallChange = Int(VScroll1.Max / 10)
Else
    VScroll1.Enabled = False
End If
End Sub

```

Eine weitere Maßnahme zur Erhöhung des Bedienkomforts ist das Vergeben von ToolTip-Texten zu den Steuerelementen. Diese vergeben Sie im Eigenschaftsfenster in der Eigenschaft *ToolTip*. Diese Eigenschaft gibt es zu den meisten Steuerelementen. Der dort eingegebene Text sollte eine stichwortartige Beschreibung der Aufgabe des Steuerelements sein (Abbildung 6.124).



Abbildung 6.124:
ToolTip für Datei-
listfeld gesetzt

Sie haben nun einen funktionierenden und komfortablen PictureBox, der in der Lage ist, die gängigsten Typen von Bilddateien anzuzeigen.

6.20 Das Steuerelement »Data«

Das Daten-Steuerelement (Data Control) ermöglicht das Navigieren durch die Datensätze einer Datenbank (z. B. einer MS-Access-Datenbank). Sein Aussehen ist über seine Eigenschaften nur gering zu beeinflussen.

6.20.1 Übung: Durch den Datenbestand einer Access-Datenbank blättern



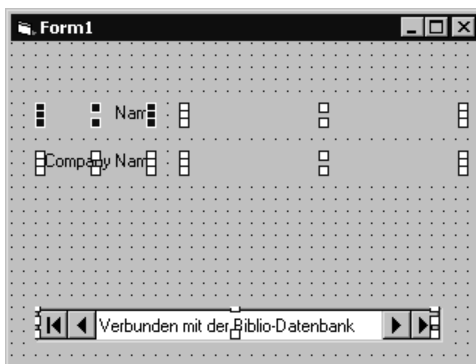
Das Daten-Steuerelement soll mit der Access-Datenbank *Biblio.Mdb* und innerhalb dieser mit der Datentabelle *Publishers* verbunden werden. Durch diesen Datenbestand muss es möglich sein, zu navigieren, das heißt datensatzweise vor- und zurückblättern etc. zu können, wobei ausschließlich die Datenfelder *Name* und *Company Name* anzuzeigen sind (Abbildung 6.125).

Abbildung 6.125:
Lauffähige Daten-
bank-Applikation



Lösung

Abbildung 6.126:
Daten-Steuer-
element und vier
Bezeichnungs-
felder (Label) im
Formular
aufgenommen



Gestalten Sie Ihr Formular so, wie in Abbildung 6.126 ersichtlich.

Damit wir auch im Daten-Steuererelement erkennen, mit welcher Datenbank wir verbunden sind, setzen wir die Eigenschaft *Caption* auf den Inhalt *Verbunden mit der Biblio-Datenbank* (Abbildung 6.126).

Um das Daten-Steuererelement mit einer im System existierenden Microsoft Access Datenbank verbinden zu können, muss die Eigenschaft *DataBaseName* versorgt werden.

Klicken Sie bei der Eigenschaft *DataBaseName* auf die Schaltfläche mit den drei Punkten, so öffnet sich der Dialog, wie in Abbildung 6.127 zu sehen, aus dem Sie die gewünschte Datenbank, in unserem Beispiel die *Biblio.Mdb*, auswählen.



Damit ist unser Daten-Steuererelement mit der Datenbank *Biblio.Mdb* verbunden, aber noch nicht mit Daten.

Da sich Daten grundsätzlich in Tabellen (Datentabellen) befinden – und von denen kann es innerhalb einer Datenbank eine ganze Menge geben (z.B. Arti-

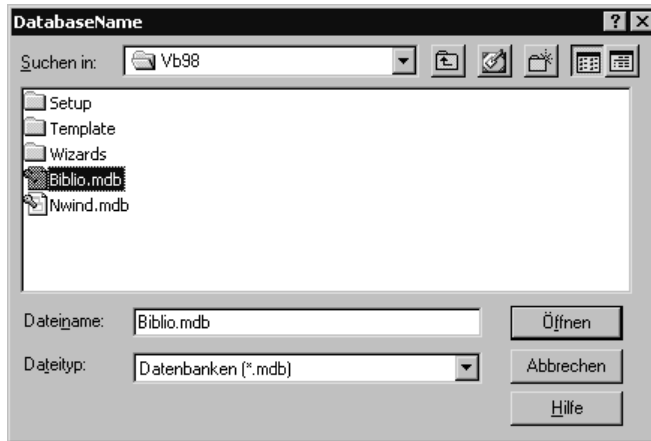


Abbildung 6.127:
Datenbank aus-
wählen

keltabelle, Kundentabelle, Lagertabelle etc.) –, ist im nächsten Schritt die gewünschte Tabelle, in unserem Fall die Tabelle *Publishers*, auszuwählen.

Alle verfügbaren Tabellen einer verbundenen Datenbank können über die Eigenschaft *RecordSource* gelistet werden. Wählen Sie hier unsere gewünschte Datentabelle *Publishers* aus.

Jetzt, nachdem wir endgültig über das Daten-Steuerelement mit der Datenbank *Biblio.Mdb* und in dieser mit der Datentabelle *Publishers* verbunden sind, werden dennoch keine Datensätze sichtbar, wenn wir nun unsere Applikation starten.



Unsere Datenausgabe-Objekte, die Bezeichnungsfelder *Label3* und *Label4* liegen ja bis jetzt noch quasi »lose« im Applikationsformular – sie sind *ungebunden*.

Damit diese einen Bezug zu unserer Datenbank herstellen können, sind sie über die Eigenschaft *DataSource* an unser Daten-Steuerelement *Data1*, das mit den *Publishers*-Daten verbunden ist, anzubinden. Dadurch werden die Bezeichnungsfelder *Label3* und *Label4* zu *gebundenen* Steuerelementen.

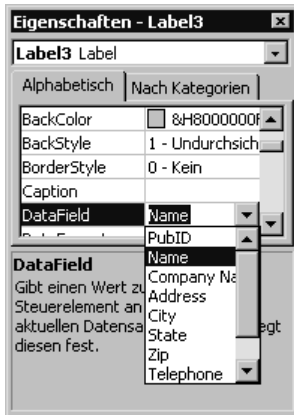
Leider werden immer noch keine Datensätze sichtbar, wenn wir jetzt unsere Applikation starten.



Zu guter Letzt ist, wie in Abbildung 6.128 zu sehen, über die Eigenschaft *DataField* eines jeden Bezeichnungsfeldes das anzuzeigende Datenfeld aus der verbundenen *Publishers*-Tabelle auszuwählen.

Starten Sie nun Ihre Applikation z.B. über F5, so können Sie, wie in Abbildung 6.125 zu sehen, durch den *Publishers*-Datenbestand der Datenbank *Biblio.Mdb* in Bezug auf die Datenfelder *Name* und *Company Name* navigieren.

Abbildung 6.128:
Datenfeld
auswählen



Die Pfeiltasten des Daten-Steuerelements bedeuten dabei (von links nach rechts) :

- ▶ Erster Datensatz – Anfang des Datenbestandes (BOF = Bottom of File)
- ▶ Vorangehender Datensatz
- ▶ Nächster Datensatz
- ▶ Letzter Datensatz – Ende des Datenbestandes (EOF = End of File)



Mit dem Daten-Steuerelement *Data* und Objekten, wie zum Beispiel Bezeichnungsfeldern (*Label*), die an das Daten-Steuerelement gebunden werden, ist es, wie in dieser Übung gezeigt, mit Visual Basic möglich, Datenbank-Applikationen ohne Programmieraufwand (keine Sourcezeile geschrieben), also nur durch das Setzen der richtigen Eigenschaften, zu entwickeln.

6.21 Das Steuerelement »DBList«

Das DBList-Steuerelement (DBList) ist ein datengebundenes Listenfeld, das automatisch von einem Feld eines zugehörigen Daten-Steuerelements gefüllt wird. Es kann ein Feld einer zugehörigen Tabelle oder eines anderen Daten-Steuerelements aktualisieren (siehe Abbildung 6.129).



6.21.1 Übung: Den Datenbestand einer Access-Datenbank listen

Das DBList-Steuerelement soll mit der Access-Datenbank *Biblio.Mdb* und innerhalb dieser mit der Datentabelle *Publishers* verbunden werden. Im Steuerelement selber ist das Datenfeld *Name* zu listen (Abbildung 6.129).

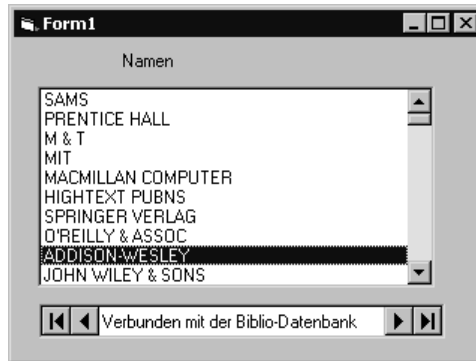


Abbildung 6.129:
Lauffähige Daten-
bank-Applikation

Lösung

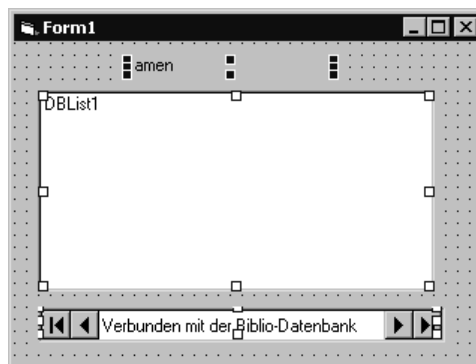


Abbildung 6.130:
Das DBList-Steuer-
element, ein
Bezeichnungsfeld
(Label) und das
Daten-Steuer-
element (Data) im
Formular
aufgenommen

Gestalten Sie Ihr Formular so, wie in Abbildung 6.130 ersichtlich, und setzen Sie die Eigenschaften des Daten-Steurelements (*Data*) (Verweise auf die Datenbank *Biblio.mdb*, Tabelle *Publishers*), wie im vorangehenden Kapitel besprochen.

Wenn Sie nun Ihre Applikation starten, werden noch keine Namen im DBListen-Steurelement angezeigt, da unser Datenausgabe-Objekt *DBList*-Steurelement *DBList1* bis jetzt noch quasi »lose« im Applikationsformular liegt – es ist *ungebunden*.



Damit dieses einen Bezug zu unserer Datenbank herstellen kann, ist es über die Eigenschaft *RowSource* an unser Daten-Steurelement *Data1*, das mit den *Publishers*-Daten verbunden ist, anzubinden. Dadurch wird das DBListen-Steurelement *DBList1* zu einem *gebundenen* Steuerelement.

Leider werden immer noch keine Namen gelistet, wenn wir unsere Applikation starten.



Zu guter Letzt ist über die Eigenschaft *ListField* des DBList-Steuerelements *DBList1* das zu listende Datenfeld (Name) aus der verbundenen Publishers-Tabelle auszuwählen.

Starten Sie nun Ihre Applikation, so werden Ihnen, wie in Abbildung 6.129 zu sehen, alle im Publishers-Datenbestand der Datenbank *Biblio.Mdb* verfügbaren Namen in einer Listbox gezeigt.



Auch mit diesem Steuerelement in Verbindung mit dem Daten-Steuerelement *Data*, ist es, wie in unserem Beispiel gezeigt, mit Visual Basic möglich, Visualisiererelemente für Datenbank-Applikationen ohne Programmieraufwand (keine Sourcezeile geschrieben), also nur durch das Setzen der richtigen Eigenschaften, zu entwickeln.

Stichwortverzeichnis

!

199

A

Abbruchkriterium 107
Abs 182
Access 339, 342
ActiveX 17
ActiveX-Steuerelemente 204, 216
Add-In-Manager 30
AddItem 161, 298, 301
ADO 17
Algebra 72
Alignment 258, 281
Alphanummerische Datentypen 43
Anführungszeichen 50
Anzeige-Element 325
API 312, 316
Application Caller 241
Arbeitsabläufe 243
Array 63
As Type 199
Asc 185
ASCII-Tabelle 89
asynchron 243, 319
Atn 182
Aufbauinterpretation 264
Aufrufparameterliste 187
Ausrichten von Steuerelementen 208
Automatische Anweisungsergänzung 34

B

Befehlsschaltfläche 235
Benutzerdefinierte Datentypen 58
Bewertungsdatei 320
Bezeichner 41
Bezeichnungsfeld 257
Biblio.Mdb 339, 343
Bildbetrachter 326
Bildfeld 307, 328
Bildformate 332
Bildlaufleisten 305

Bildschirmkoordinaten 244
Bildschirmmitte 230
bildschirmmittig 230
BOF 342
Boolean 43
BorderStyle 251, 331
Button hüpft 238
Buttongröße dynamisch 239
ByRef 189
ByVal 189

C

Call 190
Case 95
Cbool 185
Cbyte 185
Ccur 185
Cdate 185
CDbl 76, 185
Cdec 185
Cerr 185
Change 257, 332
CheckBox 294, 295
Checked 333
Cint 185
Clear 304
Client-Server 15
CLng 185
Close 128
Code anzeigen 32
Code-Fenster 34, 214
Color 228
ComboBox 296
CommandButton 235
Const 67
Cos 182
CSng 185
CStr 185
CurrentX 230
CurrentY 230
Cvar 185

D

Data 339
 DataBaseName 340
 DataField 341
 DataSource 341
 Date 49
 Dateilistbox 327
 Dateilistenfeld 255
 Dateoperationen 123
 Datenbank 339, 342
 Daten-Steuererelement 339
 Datentyp 39, 42
 Datentyp Date 49
 Datentyp String 49
 Datentypen
 – Boolean 43
 – Byte 43
 – Currency 44
 – Date 44
 – Fließkomma 43
 – Integer 43
 – Longinteger 43
 – String 43
 – Variant 44
 Datentypen im Detail
 – Variant 54
 Datumsliteral
 – Double 49
 Datumsliterale 67
 DBList 342
 DBList-Steuererelement 342
 DCOM 16
 Definitionsource 263, 266, 284
 Definitionsourcdatei 275
 Definitionszeile 187
 Deklaration 39
 DHTML 17
 Dialog Prozedur hinzufügen 194
 Dim 42
 dimensionieren 248
 Direktfenster 28
 Direktzugriffsdateien 60
 DirListBox 253
 DoEvents 128, 322
 Drive 252, 254, 255
 DriveListBox 252
 dynamisch 240, 247, 263, 270, 283

dynamische Anlage von Objekten 247, 270
 dynamische Anlage von Objekten und Instanzenbildung 281
 dynamischer String 49

E

Editionen 14
 Editor 16, 34, 279
 Eigenschaften 213, 216
 Eigenschaftenfenster 30
 Eigenschaftenliste 31
 einfügbare Objekte 204
 Einzelschrittmodus 29
 enabled 336
 End Function 199
 End Of File 127
 End Select 95
 End Sub 187
 Enterprise Edition 14
 Entscheidungsstruktur
 – If...Then-Anweisung 87
 – Select-Case-Struktur 95
 Entscheidungsstrukturen 87
 Entwicklungsumgebung 15, 19
 EOF 127, 342
 ereignisgesteuert 213
 Ereignisprozedur 214
 Ereignisse 213, 214
 – auswerten 215
 Err-Objekt 105
 Exit For 108
 Exit Function 199
 Exit Sub 105, 187
 Exp 182

F

Fakultät 112
 Fehlerbehandlung 99
 Fehlerbehandlungsroutine 99
 Feldbezeichnungen 266
 Feldnummer 273, 287
 Feldvariable 63
 FileListBox 255
 Fix 182
 For Input 126
 Form 216
 Format 130, 315, 316

Format-Funktion 173
 Formatieren von Strings 173
 Form-Ausgabeinterpreter 262
 Form-Eingabeinterpreter 281
 Formkoordinaten 234
 Formpositionierung 232
 Formular 216
 Formularfenster 35
 Frame 250, 290, 294, 295
 FreeFile 125
 freie Verschieben 260
 Funktion 133, 186, 199
 Funktion Len 52

G

Genauigkeit 46
 generieren 264, 269, 284
 generierte Programmschnellstartleiste 250
 GetPrivateProfileString 316
 GetTickCount 322
 GoTo 98
 Grafiksymbol 137
 Grenzwert 107
 Gruppe hinzufügen 208
 Gültigkeitsbereich 187
 – Private 188
 – Public 187
 Gültigkeitsbereich einer Variablen 69

H

Haltemodus 27
 Haltepunkt setzen 26
 Height 331
 Hex 185
 Hierarchische Beziehung 290
 Hilfedatei 141
 Hilfekontext 141

I

Identifizier 316
 Image 325, 327, 337
 implizite Deklaration 40
 Index 63, 247
 Indexwerte 63
 indiziert 247
 Indizierung 288
 InputBox 146
 Instanzen 247

Instr-Funktion 164
 Int 182
 Integerdivision 74
 integrierten Steuerelemente 204
 internationalisieren 264
 Interpretersource 263, 267, 273, 285
 Interpretersourcedatei 274
 interpretieren 263, 269, 287
 Interval 310
 Is 96
 IsMissing 189

K

Kaskaden-Menü 224
 Kategorien 31
 Kernel32 322
 KeyPress 278, 288, 313
 Klasse 213
 Kombinationsfeld 296
 Komponenten 203
 Konstanten 67
 Kontrollkästchen 294, 305
 Konvertierungsfunktion 73, 76
 Kundenindividuell 264

L

Label 257
 Ländereinstellungen 49
 LargeChange 306, 337
 Laufwerkslistbox 330
 Laufwerkswechsel 252
 Laufzeit 263, 270, 282
 Laufzeitbibliothek 133
 Laufzeitfehler 50, 58, 65, 99
 LBound-Funktion 66
 LCase-Funktion 156
 Left-Funktion 157
 Len-Funktion 154
 Line 228
 Line Input 123
 Listbox 299
 Listenfeld 299
 ListField 344
 ListIndex 302, 336
 Literal 67
 LoadPicture 333
 Log 182
 logische Operatoren 80

- And 81
- Eqv 81
- Not 81
- Or 81
- Xor 81

lokale Variable 70
 Lokalfenster 28
 LTrim-Funktion 155

M

Masterobjekt 247, 273, 287
 Max 306, 337
 MaxLength 288, 313
 MDI 16
 Me 335
 mehrsprachenfähig 264, 284
 Menü

- Ansicht 23
- Ausführen 29
- Bearbeiten 22
- Datei 20, 21
- Debuggen 25
- Extras 29
- Extras->Optionen 29
- Fenster 30
- Format 25
- Projekt 23

 Menübefehl 329

- Eigenschaften von Projekt1 25
- Komponenten 24
- Verweise... 23

 Menü-Code 226
 Menü-Editor 29, 220, 329
 Menüpunkt

- Add-Ins 30

 Menüs deaktivieren 226
 Menütypen 220
 Methoden 213, 216
 Mid-Funktion 158
 Min 306, 337
 Modulkomponenten 289
 Modulo 73
 MouseDown 230, 309
 MouseMove 227, 238, 261
 Movable 219
 MsgBox 135
 MultiLine 280
 MultiSelect 305

Multitaskingfähig 319
 Multithreading DLLs 17

N

Nachkommastelle 176
 Namenskonventionen 47
 Neupositionierung 261
 New 42
 Next 108
 Nothing 71
 Now 130, 316
 numerisch geprüft 288
 numerische Datentypen 43
 numerische Eingabeprüfung 278

O

Objekt 214
 Objekt anzeigen 32
 Objektabstand 212
 Objektkatalog 69
 Objekt-ListBox 31
 Oct 185
 Öffentliche Variable 70, 71
 OLE 17
 On Error 99
 Online-Hilfe 123, 133
 Open 123
 Operator 72

- & 50
- Operator Priorität 74

 Operators And 81
 Operators Eqv 82
 Operators Not 82
 Operators Or 82
 Operators XOr 82
 Option Explicit 40
 Optional 189
 OptionButton 290, 294
 Optionsfeld 290
 Ordner 253

P

Paket- und Weitergabeassistent 33
 parallel 324
 Parameter Schaltflächen 136
 Parameterliste 188
 PasswordChar 276, 277
 Passwort 275

Path 254, 256, 335
 Pattern 332
 Picture 308, 326
 PictureBox 307, 327, 337
 PictureBox 326
 PrintForm 219
 Private 42, 187
 Private Variable 70
 Professional Edition 14
 Programmieraufwand 263, 289
 Programmiersprache 19
 Programmschleifen 107
 Programmschnellstartleiste 242
 Projekt speichern 22
 Projekt/Bibliotheks-Liste 69
 Projektdatei speichern 22
 Projekt-Explorer 32, 33
 Prozedur 133, 186, 187
 – hinzufügen 194
 Public 42, 71, 187
 Publishers 339, 343

Q

QBColor 228
 QuickInfo 35, 36, 237

R

Rahmen 290
 Rahmenstil 250
 Rahmentext 250
 Randomize 183
 Rechenfunktion 182
 Rechenoperatoren 72
 RecordSource 341
 ReDim 42
 referenziert 287
 Referenzierung 285
 Referenznummer 267, 270, 285
 Registerkarte Index 134
 Registerkarte Inhalt 133
 Registerkarte Suchen 134
 RemoveItem 303
 Resize 241
 Right-Funktion 158
 Rnd 182, 183
 RTrim-Funktion 155

S

Schiebebalken 337
 Schleifenstruktur
 – Do...Loop-Schleife 117
 – For...Next-Schleife 107
 ScrollBars 281, 305, 337
 Select Case 95
 Sgn 182
 SHELL 249
 Sin 134, 182
 SmallChange 306, 337
 sorted 304, 315
 Spezialobjekt 335
 Sqr 182
 Standard Edition 14
 Standard-Befehlsschaltfläche 138
 Standard-EXE 21
 Startleistendatei 244
 Start-Projekt 33
 Static 42, 188
 statischen String 50
 Step 108
 Steuerelement 15, 214
 stoppen 317
 Stoppuhr 317
 Stretch 326, 327, 333
 Str-Funktion 172
 Style 305
 Sub 187
 symbolische Konstanten 67
 Symbolleisten 30
 Synchronisation 319

T

Tan 182
 Tastaturkürzel 236
 Tausendertrennzeichen 176
 Termine 311
 Terminplaner 310
 TextBox 275
 Timer 181, 309, 315, 322
 Titel 140
 ToolTip 317, 339
 ToolTipText 237, 238
 Transaktionen 243
 Trim-Funktion 155
 Type-Anweisung 59
 Typprüfung 43

U

UBound-Funktion 66
 UCase-Funktion 156
 Überwachung hinzufügen 28
 Überwachungsfenster 28
 Uhrzeit anzeigen 309
 Umwandlungsfunktion 185
 ungebunden 341, 343
 UNLOAD 249
 Untertyp Empty 55
 Untertypen 54
 Until 117

V

Val-Funktion 171
 Value 292, 337
 variabel 263
 Variable 39
 Variablendeklaration erforderlich 40
 Variablenname 41
 Variant 43
 VarType-Funktion 54
 VbAbort 141
 VbAbortRetryIgnore 137
 VbApplicationModal 139
 VbBinaryCompare 164
 VbCancel 141
 VbCritical 138
 VbDatabaseCompare 165
 VbDefaultButton1 139
 VbDefaultButton2 139
 VbDefaultButton3 139
 VbDefaultButton4 139
 VbExclamation 138
 VbIgnore 141
 VbInformation 138
 VbMsgBoxHelpButton 139
 VbMsgBoxRight 140
 VbMsgBoxRtlReading 140
 VbMsgBoxSetForeground 139
 VbNo 141
 VbOK 141

VbOKCancel 137
 VbOKOnly 137
 VbQuestion 138
 VbRetry 141
 VbRetryCancel 137
 VbSystemModal 139
 VbTextCompare 165
 VbYes 141
 VbYesNo 137
 VbYesNoCancel 137
 Verbindungs-Designer 18
 Vergleichsoperatoren 77
 Verkettung 153
 Verzeichnislistbox 330
 Verzeichnislistenfeld 253
 Visible 236, 334
 vordefinierte Konstanten 68
 vordefinierte Zeichenkettenkonstanten 53
 Vorgabewert 146

W

Wahrheitstabelle 81
 Werkzeugsammlung 35, 203
 While 117
 Width 331
 Windowsboot 243
 WindowState 335
 WindowStyle 245, 249
 WritePrivateProfileString 316

Z

Zähler 107
 Zeichenketten 49
 Zeichenkettenkonstante vbCrLf 53
 Zeichenkettenliteralen 67
 Zeitabschnitte 319
 Zeitgeber 309
 Zeitraster 319
 zentrieren 231, 257
 zusätzlichen Steuerelemente 204
 Zuweisungsoperator 48