PEARSON Education

ADDISON-WESLEY

Dirk Abels

Visual Basic 6 lernen

Anfangen, anwenden, verstehen

ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Bonn • Reading, Massachusetts • Menlo Park, California
New York • Harlow, England • Don Mills, Ontario
Sydney • Mexico City • Madrid • Amsterdam

3 Die Programmentwicklung

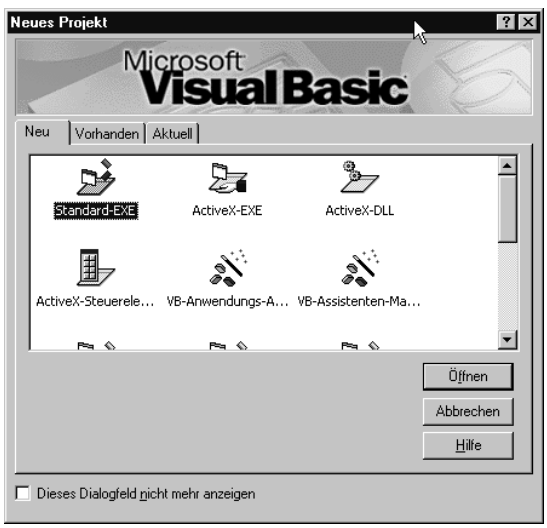

Abbildung 3.1: Auswahl nach dem Programmstart

3.1 Das einfachste Visual Basic-Programm

Ein Programm ohne Programmieraufwand

Nach dem Aufruf des Start-Menüs von Windows 95 und der Menüpunkte PROGRAMME | VISUAL BASIC 6.0 kann durch Anklicken von VISUAL BASIC 6.0 die Entwicklungsumgebung gestartet werden. In dem Dialog NEUES PROJEKT WÄHLEN Sie die Option STANDARD-EXE aus. Jetzt ist die Grundlage für das erste Programm erstellt worden. Die Entwicklungsumgebung enthält ein Grundprogramm. Durch Drücken der Taste F5 oder über die Menübefehle RUN | START wird das aktuelle Programm gestartet. Dieses einfachste Visual Basic-Programm enthält ein leeres Formularfenster, da bis zu diesem Zeitpunkt noch keine Programmierung erfolgt ist.

Dennoch ist das Programm in diesem Stadium bereits leistungsfähig. Das Fenster kann vergrößert oder verkleinert, auf der Oberfläche verschoben und sogar beendet werden; es entspricht dem Windows-Standard. Die genannten Befehle können durch Anklicken des Symbols in der linken oberen Ecke ausgeführt werden. Wenn man das Programm jedoch über einen Button beenden will, muß dieser Befehl `(end)` explizit programmiert werden.

3.2 Grundlegende Programmstrukturen

3.2.1 Kurzüberblick

Dieses Kapitel erläutert die verschiedenen Variablentypen und Konstanten. Es wird gezeigt, welche Wertebereiche Variablen einnehmen, und erklärt, wie man für eine bestimmte Aufgabe die richtige Variable findet. Weiterhin wird der Einsatzbereich der Konstanten erklärt.

3.2.2 Variablen

Variablen sind über einen Namen gekennzeichnete und ansprechbare Größen. Sie bestehen aus zwei Elementen, dem Variablennamen und dem Eigenwert der Variablen. Der Name der Variablen kann während der Programmlaufzeit nicht verändert werden, der Eigenwert dagegen sehr wohl. Variablen werden benötigt, um z.B. Ergebnisse von Berechnungen oder Eingaben von Benutzern zu speichern und zu verarbeiten. In Visual Basic ist es nicht zwingend erforderlich, daß eine Variable deklariert[1] wird. Es wird jedoch empfohlen, im Menü EXTRAS OPTIONEN... auf dem Tabulator EDITOR die Option VARIABLENDEKLARATION ERFORDERLICH zu aktivieren, damit grundsätzlich alle Variablen deklariert werden müssen. Ansonsten können Programmfehler entstehen, deren Ursachen nur sehr schwer zu finden sind (Tabelle 3.1).

Das folgende Beispiel soll die Summe aller Zahlen von 1 bis 100 addieren und das Ergebnis an die aufrufende Funktion zurückliefern. Die Unterschiede des Programmcodes (`Erg` und `Erk`) in den Spalten „richtig“ und „falsch“ sind grau hinterlegt.

1. Vor der Benutzung wird dem Programm bekanntgegeben, um welchen Variablentyp (Zeichenkette, Zahl usw.) es sich handelt.

	richtig	falsch

Programm	`function Summe(Zahl as integer) as integer`
	
	`dim i as integer`
	`dim Erg as integer`
	
	`Erg = Erg + i`
	`for i = 0 to Zahl`
	`Erg = Erg + i`
	`next i`
	`Summe = Erg`
	`end Function`
		`function Summe(Zahl as integer) as integer`
		
		`dim i as integer`
		`dim Erg as integer`
		
		`Erg = Erk + i`
		`for i = 0 to Zahl`
		`Erg = Erk + i`
		`next i`
		`Summe = Erg`
		`end Function`

Übergabewert	Zahl = 100	Zahl = 100
Ausgabe	Summe = 5050	Summe = 100

Tabelle 3.1: Möglicher Fehler aufgrund fehlender Deklarationen von Variablen

Das Programm wird, wenn nicht explizit eine Deklaration der Variablen gefordert ist, keine Fehlermeldung ausgeben. Der Unterschied besteht nur darin, daß bei der Version „Falsch“ mit einem falschen Ergebnis weitergerechnet wird. In diesem kleinen Beispiel ist der Fehler relativ einfach zu lokalisieren. In sehr umfangreichen Programmen dagegen ist die Suche sehr zeitaufwendig. Unter Umständen sucht man den Fehler sogar an der falschen Stelle.

Variablentypen

In Visual Basic stehen für die Programmierung folgende Variablentypen zur Verfügung:

Typ	Wertebereich	Beschreibung
Byte	0 bis 255 1 Byte	*Byte* sind ganze Zahlen, die hauptsächlich bei Berechnungen und Ausgaben verwendet werden, die aus den ASCII-Werten einzelner Zeichen aus dem ASCII-Zeichensatz bestehen (siehe Anhang).
Boolean	True, False 2 Byte	Signalisiert den Zustand Boolscher Ausdrücke oder Variablen. Achtung: In Visual Basic entspricht der Wert `-1` dem Zustand *True*/*Wahr* und der Wert 0 dem Zustand *False*/*Falsch*.
Integer	–32768 bis +32767 2 Byte	Die Variablentypen *Integer* und *Long* sind Ganzzahl-Variablentypen. Sie werden dort eingesetzt, wo keine Nachkommastellen benötigt werden.

Typ	Wertebereich	Beschreibung
Long	–2`147`483`648 bis 2`147`483`647 4 Byte	Die beiden Variablentypen unterscheiden sich darin, daß der Wertebereich von *Long* deutlich größer als der von Integer ist. Eine Variable vom Typ *Long* nimmt im Vergleich zu einer vom Typ *Integer* auch doppelt soviel Speicherplatz ein.
Single	Im negativen Bereich: 3`3,402823*10^{38} bis –1.401298*10^{-45} Im positiven Bereich: 1,401298*10^{-45} bis 3,402823*10^{38} 4 Byte	*Single* und *Double* sind Fließkommazahlen, die sich darin unterscheiden, daß *Double* einen weitaus größeren Wertebereich hat als *Single*, allerdings doppelt soviel Speicher benötigt wie eine Variable vom Typ *Single*. Diese Variablentypen werden bei Berechnungen eingesetzt, die sehr genau sein müssen.
Double	Im negativen Bereich: –1.79769313486232*10^{308} bis –4.94065645841247*10^{-324} Im positiven Bereich: 4.94065645841247*10^{-324} bis 1.79769313486232*10^{308} 8 Byte	
Currency	–922`337`203`685`477.5808 bis 922`337`203`685`477.5807 8 Byte	*Currency* ist eine Mischung aus einer Ganz- und einer Fließkommazahl. Damit lassen sich Berechnungen bis zur vierten Stelle nach dem Komma durchführen.
Date	Datum 1. Januar 100 bis 31. Dezember 9999 Uhrzeit 00:00 bis 23:59:59 8 Byte	Wird benutzt für Datums- und Uhrzeitangaben, wobei die im Wertebereich angegebenen Begrenzungen gelten.
String	10 Byte + 2 Byte pro Zeichen	Zeichenketten bestehen aus der Aneinanderreihung einzelner Zeichen aus dem ASCII-Zeichensatz (siehe ASCII-Tabellen im Anhang). Seit der Visual Basic-Version 4.0 ist die Länge einer Zeichenkette nur noch durch den Hauptspeicher begrenzt (2.147.483.647 Zeichen).
Object	4 Byte	Die Variable speichert einen Verweis auf ein Objekt. Es kann sich hierbei um einen Button oder ein Formular handeln. Sie wird eingesetzt, um Zugriffe auf Objekte übersichtlicher zu gestalten.
Variant	Bei Numerischen Werten: 16 Byte Bei Zeichenketten: 22 Byte + 2 Byte pro Zeichen	*Variant* ist ein *Default*-Variablentyp, der jeden beliebigen Typ aus der obigen Tabelle annehmen kann. Variablen werden automatisch konvertiert. Dieser Variablentyp wird dort eingesetzt, wo sich der Typ einer Variable bei der Programmausführung ändern kann.

Tabelle 3.2: Variablentypen und ihre Beschreibung

Im folgenden sollen einige Beispiele zur Deklaration von Variablen gezeigt werden.

Code	Beschreibung
`Dim i As Integer`	Deklariert `i` als Ganzzahl
`Dim Zeichenkette As String`	Deklariert *Zeichenkette* als Zeichenkette
`Dim abcdefg`	Kein Variablentyp angegeben, also wird der allgemeine Variablentyp *Variant* verwendet

Tabelle 3.3: Beispiele zur Variablendeklaration

Alle Deklarationen der Tabelle 3.3 sind erlaubt, auch Deklarationen ohne Angabe des Variablentyps – Visual Basic sucht für die Variable automatisch den richtigen Variablentyp heraus. Jedoch ist diese Art von Deklaration nicht zu empfehlen, da sich die Suche negativ auf die Geschwindigkeit der Programmabarbeitung auswirkt.

Es gibt eine zusätzliche Kurzschreibweise, um den Typ einer Variablen festzulegen. In Tabelle 3.4 sind Zeichen aufgelistet, die eine Variable eindeutig einem bestimmten Variablentyp zuordnen.

Zeichen	Variablentyp
%	Integer
&	Long
!	Single
#	Double
@	Currency
$	String

Tabelle 3.4: Symbole für die Variablendeklaration

Wenn man bei der Deklaration der Variablen die in Tabelle 3.6 vorgestellten Zeichen für die Bestimmung des Variablentyps verwendet, ist der Programmcode für den Programmierer, der sich zunächst in den Programmfluß hineindenken muß, sehr viel einfacher zu verstehen, da er den Variablentyp bereits an der Namensgebung der Variablen erkennt.

Code	Beschreibung
`Dim i%`	Deklariert `i` als Ganzzahl
`Dim Zeichenkette$`	Deklariert *Zeichenkette* als Zeichenkette
`Dim Wert% As Integer`	Keine gültige Deklaration der Variablen, da sonst
`Dim Wert& As Integer`	die Variablen doppelt deklariert wären.

Tabelle 3.5: Beispiele zur Variablendeklaration durch den Variablennamen

Der allgemeine Variablentyp *Variant*

In Tabelle 4.2 wurde eine Variable ohne Typangabe deklariert. Dieser Variablen wird automatisch der Variablentyp *Variant* zugewiesen. Mit den so deklarierten Variablen lassen sich alle anderen Datentypen speichern. Diese Variablen sind deshalb sehr flexibel und natürlich auch sehr bequem zu handhaben. Das folgende Beispiel soll die Einfachheit dieser Variablentypen veranschaulichen.

Es soll eine Funktion geschrieben werden, die das Quadrat einer Variablen berechnet.

```
Function Quadrat(Satz As Variant) As Variant
   Quadrat = Satz * Satz
End Function

Sub Beispiel_1()
   Dim Satz$
   Dim Zahl%
   Dim Komma!

   Satz$ = "25 DM"
   Zahl% = 20
   Komma! = 4.25

   Zahl% = Quadrat(Zahl%)
   Debug.Print "Zahl = " & Zahl%
   Komma! = Quadrat(Komma!)
   Debug.Print "Komma = " & Komma!
   Satz$ = Quadrat(Satz$)
   Debug.Print "Satz = " & Satz$
End Sub
```

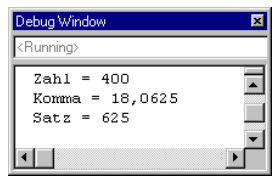

Abbildung 3.2: Ausgabe des Variant-Beispiels in das Debugfenster

In Abbildung 3.3 ist die Ausgabe des kurzen Beispielprogrammes im Debugfenster dargestellt. In der Funktion *Quadrat* wurde keine explizite Typkonvertierung programmiert. Allerdings multipliziert sie den Zahlenanteil (die 25) des *String* (25 DM) mit sich selbst und liefert das Ergebnis wieder im richtigen Datentyp zurück. Der Stringanteil (DM) geht verloren, weil intern eine Typkonvertierung durchgeführt wird.

Das folgende Beispielprogramm liefert das gleiche Ergebnis wie das vorherige Beispiel, jedoch arbeitet die Funktion mit dem Variablentyp *Long*.

```
Function Quadrat_2(Satz As Long) As Long
    Quadrat_2 = Satz * Satz
End Function

Sub Beispiel_2()
   Dim Satz$
   Dim Zahl%
   Dim Komma!

   Satz$ = "25 DM"
   Zahl% = 20
   Komma! = 4.25

   Zahl% = Quadrat_2(Zahl%)
   Debug.Print "Zahl = " & Zahl%
   Komma! = CSng(Quadrat_2(CLng(Komma!)))
   Debug.Print "Komma = " & Komma!
   Satz$ = CStr(Quadrat_2(CLng(Satz$)))
   Debug.Print "Satz = " & Satz$
End Sub
```

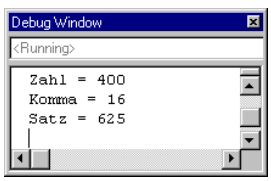

Abbildung 3.3: Ausgabe der zweiten Programmlösung zu Variant in das Debugfenster

Beim Vergleich der beiden Programme fällt zum einen auf, daß das Programm mit dem Variablentyp *Variant* weniger komplex ist als das mit dem Variablentyp *Long*. Des weiteren ist die Berechnung des Quadrats einer Kommazahl im zweiten Beispiel falsch, denn bei der Übergabe der Variablen an die Funktion *Quadrat* wird der Nachkommateil einfach abgeschnitten.

3.2.3 Konstanten

Konstanten bestehen wie die Variablen aus zwei Einheiten, dem *Konstantennamen* und dem *Eigenwert*. Im Gegensatz zu den Variablen kann weder der Eigenname noch der Wert der Konstanten während der Laufzeit verändert werden. Dies ist auch der Zweck ihrer Verwen-

dung: Wenn im Programm Variablen eingesetzt werden, deren Werte nicht verändert werden, und die Werte auch schon zur Zeit der Programmierung bekannt sind, handelt es sich bei diesen Werten um Konstanten. Eine Konstante kann von jedem beliebigen Variablentyp sein, denn jede Variable kann als Konstante definiert werden, indem vor den Ausdruck der Variablendeklaration der Befehl *const* gestellt wird.

```
Const Maximum = 181
Const Maximum As Long = 181
Const Maximum As Double = 181
Const Pi As Double = 3.1415
```

Es ist sogar folgender Ausdruck möglich, wobei Pi (π) dann nur der Wert 3 zugewiesen wird.

```
Const Pi As Integer = 3.1415
```

Eine typische Konstante für ein Programm ist z.B. die Versionsnummer. Sie darf während der Programmlaufzeit nicht verändert werden, muß aber immer wieder vom Programmierer angepaßt werden.

Weitere Beispiele für Konstanten sind:

- Umrechnungsfaktoren für physikalische Masse
- Mathematische Konstanten (z.B. p, Eulersche Zahl)
- Temporäre Dateinamen, die für Auslagerungszwecke benutzt werden
- Maximum- und Minimumwerte, die für bestimmte Berechnungen nicht über- oder unterschritten werden dürfen

Das folgende Programm verwendet eine Konstante `Maximum`, um zu prüfen, ob die Zahl, deren Quadrat berechnet werden soll, kleiner ist als der größte erlaubte Wert.

```
Function Quadrat(Zahl As integer) As Integer
   If ABS(Zahl) <= Maximum Then
      Quadrat = Zahl * Zahl
   Else
      Quadrat = -1
   Endif
End Function
```

3.2.4 Mehrdimensionale Felder

Felder sind Listen von Variablen. Betrachtet man ein Zeichen als einen Datentyp, so ist die Zeichenkette vom Typ *String* ein Feld von Zeichen. Demnach wäre eine Variable *Satz*, die als

```
Dim Satz As Char[100]
```

deklariert wird, eine Zeichenkette mit 100 Zeichen. Felder werden also immer dort eingesetzt, wo mehrere Informationen vom gleichen Datentyp gespeichert werden. Die Felder finden beispielsweise Anwendung in der Vektorrechnung.

Statische Felder

Wenn die Größe eines Feldes konstant bleibt, spricht man von *statischen* Feldern. Der Inhalt der einzelnen Feldvariablen kann jederzeit geändert werden, aber die Anzahl der Felder, die zu der Variablen gehören, ist konstant. Im folgenden wird eine Variable *Woche* deklariert, die alle Wochentage enthält.

```
Dim Woche(7)
Woche = Array("Sonntag", "Montag", "Dienstag", _
              "Mittwoch", "Donnerstag", "Freitag", _
              "Samstag")
```

Die Variable *Woche* ist nun ein Feld und enthält sieben Werte vom Typ *String*. Alle Werte sind bei der Deklaration initialisiert worden. Ausgelesen wird jeder Wert über seine Position in dem Feld.

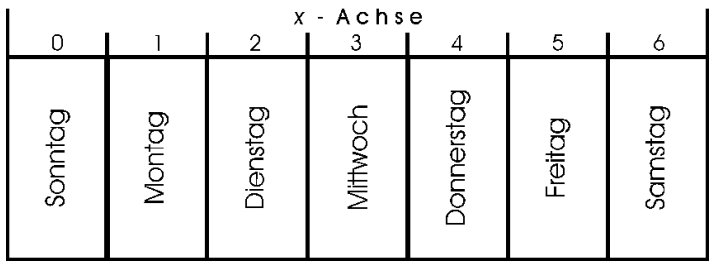

Abbildung 3.4: Eindimensionales Feld „Woche“

Wie in Abbildung 3.4 zu sehen ist, sind die Variablen aneinandergereiht, und jeder Position ist eine eindeutige Nummer zugeordnet. Folgende Programmzeile weist einer Variablen den dritten Wochentag, den Dienstag, zu:

```
Dim Tag As String
Tag = Woche(3)
```

Wenn zur Programmierzeit der Inhalt der Variablen noch nicht bekannt ist, sieht die Deklaration eines Feldes wie folgt aus:

```
Dim Woche(7) As String
```

Der Inhalt aller Felder kann jederzeit über den Zugriff des Index geändert werden.

```
Woche(2) = "Monday"
```

Mehrdimensionale Felder

In Abbildung 3.4 wurde ein eindimensionales Feld dargestellt. In dem Beispiel mit dem Feld *Woche* wurde allerdings die ganze Zeit auf einem zweidimensionalen Feld gearbeitet, wenn man davon ausgeht, daß ein String schon ein eindimensionales Feld ist. In Abbildung 3.5 ist nun ein zweidimensionales Feld dargestellt, mit dem zugehörigen Index für jede Zelle.

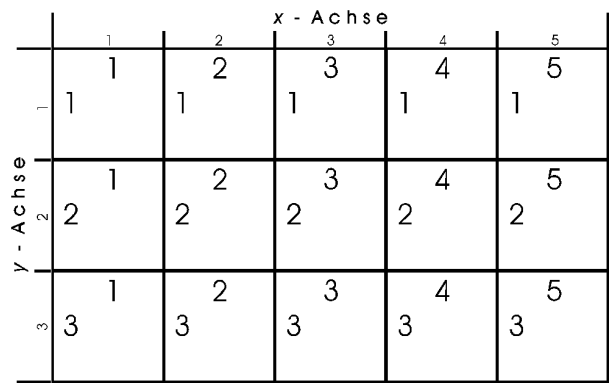

Abbildung 3.5: Zweidimensionales Feld

Abbildung 3.5 zeigt, daß nun zwei Werte nötig sind, um ein Feld eindeutig zu identifizieren: zum einen der Index der horizontalen, der *x*-Achse, und zum anderen der Index der vertikalen, der *y*-Achse. Die Deklaration einer zweidimensionalen Feldvariablen sieht wie folgt aus:

```
Dim Feld(4, 5) As Integer
```

Der Zugriff auf das dritte Feld horizontal und das zweite Feld vertikal erfolgt über den Befehl:

```
Feld(4, 5)
```

Die nächste Steigerung ist nun das dreidimensionale Feld. Es ist die letzte Stufe, die sich noch recht übersichtlich darstellen läßt (Abbildung 3.6).

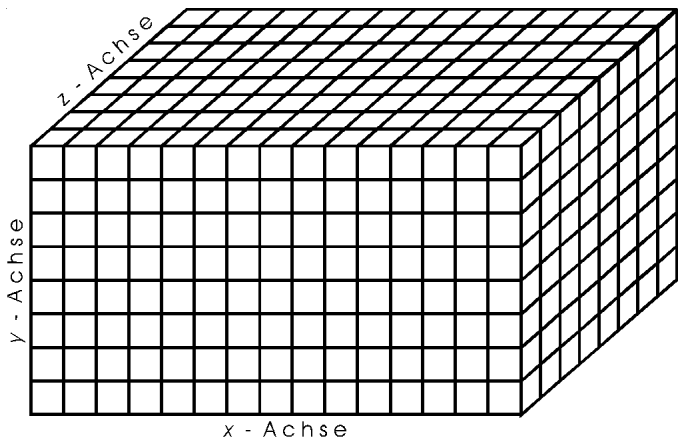

Abbildung 3.6: Darstellung eines dreidimensionalen Feldes

Auch hier erhöht sich nun wieder die Komplexität der Parameter, um auf eine bestimmte Zelle zugreifen zu können. Die Komplexität der Felder läßt sich noch weiter steigern, jedoch wird es mit jeder neuen Komplexitätsgröße schwieriger, sich diese Datenstruktur vorzustellen.

Das folgende Beispiel initialisiert ein zweidimensionales Feld der Größe 5 x 3 mit den Produkten der Indexwerte.

```
Dim i, j As Integer
Dim Feld(5, 3) As Integer

for i= 0 to 4
   for j = 0 to 2
      Feld(i, j) = i * j
   next j
next i
```

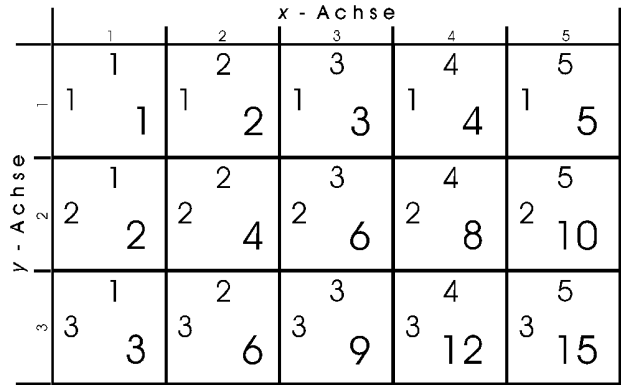

Abbildung 3.7: Zweidimensionales Feld mit berechneten Feldwerten

Benutzerdefinierter Indexbereich

Bei den bisherigen Felddeklarationen wurde die Indexnumerierung von Visual Basic vorgegeben. Es ist auch möglich, den eigenen Indexbereich vorzugeben. Wenn ein eindimensionales Feld deklariert wird, das 101 Werte enthalten soll, so verläuft der Index von 0 bis 100. Wenn jedoch ein Index der Feldvariablen von –50 bis +50 nötig ist, so muß diese wie folgt deklariert werden:

```
Dim Feld(-50 to 50)
```

Dies funktioniert ebenfalls bei mehrdimensionalen Feldern, deren Indexbereich vom Programmierer festgelegt werden muß.

```
Dim Feld(-3 To 20, 30 To 32) As Integer
```

Dynamische Felder

Bisher sind wir immer davon ausgegangen, daß die Feldgröße, also die Anzahl der Elemente, die eine Feldvariable enthalten soll, bekannt ist. Es gibt aber auch Aufgaben, bei denen die Feldgröße nicht bekannt ist. Ein Beispiel hierfür ist das Rechnen mit Vektoren und Matrizen. Die Deklaration der dynamischen Feldvariablen erfolgt wie die Deklaration der statischen Feldvariablen mit dem Befehl `Dim` und optional mit dem Variablentyp. Im Unterschied zur statischen Felddeklaration wird keine Größe des Feldes angegeben.

```
Dim Vektor() As Double
Dim Matrix() As Double
Dim Hyper()
```

Die Definition der Größe des Feldes erfolgt erst im Programm, wenn die Feldvariable gebraucht wird. Diese neue Dimensionierung der Variablen erfolgt über den Befehl `ReDim`. Er wird wie der Befehl `Dim` zur Felddeklaration verwendet, jedoch immer ohne Variablentyp.

```
ReDim Vektor(10)
ReDim Matrix(4, 3)
ReDim Hyper(2, 2, 2, 2, 2, 2, 2, 2)
ReDim Hyper(2, 2, 2, 2, 2, 2, 2, 2) As Integer 'Ist nicht möglich
```

Die Feldvariable *Vektor* ist ein eindimensionales Feld mit zehn Elementen vom Typ *Double*. *Matrix* ist schon ein zweidimensionales Feld vom Typ *Double*, enthält also

4 x 3 = 12

Variablen vom Typ *Double*. Die Variable *Hyper* ist eine achtdimensionale Feldvariable vom Typ *Variant* und enthält

2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 = 256

Variablen. Die Werte, die die Feldgröße definieren, können selbst wieder Variablen sein.

```
Dim Anzahl As Integer
Dim Feld()
   ...
Anzahl = 20
   ...
ReDim Feld(Anzahl)
```

Nach der `ReDim`-Anweisung können die Größe eines Feldes und die Dimensionierung jederzeit geändert werden. Wenn sich die Größe des Feldes während der Programmlaufzeit verändern muß, ist auch dieses mit der `ReDim`-Anweisung zu lösen (zum Ablauf sei auf die oben beschriebenen Beispiele verwiesen).

```
Dim Feld()         'Deklaration der Variablen
   ...
ReDim Feld(3)      'Erstes Festlegen der Feldgröße
   ...
ReDim Feld(5)      'n-tes Festlegen der Feldgröße
```

Bei der Neudimensionierung von dynamischen Feldern kann die Feldgröße jederzeit geändert werden, jedoch nicht die Anzahl der Dimensionen. Die folgende Programmzeile erzeugt aus dem dynamischen Feld

```
Dim Feld()         'Deklaration der Variablen
```

ein mehrdimensionales Feld:

```
ReDim Feld(4, 2)   'm-tes Festlegen der Feldgröße
```

Nachdem die Anzahl der Dimensionen eines dynamischen Feldes einmal festgelegt wurde, kann diese nicht mehr geändert werden.

Es ist bei der Redeklaration eines Feldes darauf zu achten, daß der Inhalt der Felder verlorengeht, wenn die Änderung der Felddimension wie oben durchgeführt wird. Um den Inhalt der einzelnen Feldwerte bei der Größenänderung der Feldvariablen beizubehalten, muß der Befehl `Preserve` wie folgt verwendet werden:

```
Dim Feld()         'Deklaration der Variablen
   ...
ReDim Feld(3)      'Erstes Festlegen der Feldgröße
   ...
ReDim Preserve Feld(5) 'n-tes Festlegen der Feldgröße
```

Löschen des Feldinhalts

Um alle Felder einer Feldvariablen zu löschen, gibt es unter Visual Basic den Befehl `Erase`.

```
Erase Feld
```

`Erase` gibt bei dynamischen Feldern den Speicher der Feldvariablen wieder frei, d.h., um einem Feld wieder einen Wert zuzuweisen, muß die Feldgröße der Variablen neu mit `ReDim` dimensioniert werden. Bei statischen Feldern hängt der Inhalt der Feldvariablen vom Variablentyp ab (Tabelle 4.6).

Variablentyp	Neuer Wert
Byte	0
Boolean	False
Integer, Long	0
Single, Double	0
Currency	0
Date	00:00:00
String	""
Objekt	Nothing
Variant	Empty

Tabelle 3.6: Inhalt der Feldvariablen nach dem Löschen mit Erase

Ermitteln des Indexbereiches eines Feldes

Um bei Feldern den kleinsten und größten erlaubten Index zu ermitteln, gibt es die Funktionen *LBound* (untere Grenze) und *UBound* (obere Grenze).

```
Min = LBound(Feld)
Max = UBound(Feld)
```

Bei mehrdimensionalen Feldern muß die Dimension, deren Grenzen bestimmt werden sollen, auch in der Parameterliste angegeben werden.

```
Min = LBound(Feld, 2)
Max = UBound(Feld, 2)
```

Es gibt leider noch keinen Befehl, mit dem sich ermitteln läßt, wie viele Dimensionen ein Feld hat.

Eigenheiten der Felder vom Typ *Variant*

In Abschnitt „Der allgemeine Variablentyp Variant“ stellten wir fest, daß der Variablentyp *Variant* alle anderen Variablentypen darstellen kann. Diese Aussage trifft auch auf jedes Feld einer Feldvariablen vom Typ *Variant* zu. Ein Element dieser Feldvariablen enthält eine Zahl und ein anderes Feld, einen String.

```
Dim Feld(2)

Feld(0) = "Dies ist eine Zeichenkette"
Feld(1) = 3.1415
Debug.Print Feld(0)
Debug.Print "PI = " & Feld(1)
   ...
Feld(1) = Feld(1) * 2
Feld(0) = Feld(0) & " die verlängert wurde"
Debug.Print Feld(0)
Debug.Print "2 * PI = " & Feld(1)
```

Das in das Debugfenster geschriebene Ergebnis sieht folgendermaßen aus:

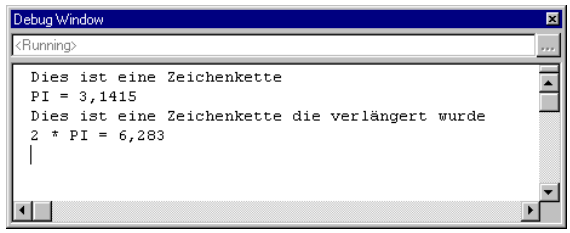

Abbildung 3.8: Ausgabe des Feldbeispiels vom Typ „Variant“

Wie in Abbildung 3.8 zu sehen ist, können somit verschiedene Variablentypen in einer Feldvariablen enthalten sein.

3.2.5 Selbstdefinierte Datentypen

Trotz der Vielfalt an angebotenen Datentypen ist es dennoch bei manchen Anwendungen nötig, eigene Datentypen zu entwickeln. Ein wichtiger Datentyp der Mathematik, der nicht von Visual Basic unterstützt wird, ist der Datentyp *Koordinate*. Um in einem Koordinatensystem mit zwei Achsen die Position eines Punkts genau festzulegen, sind zwei Angaben notwendig: die x-Koordinate und die y-Koordinate (siehe Abbildunb 3.9).

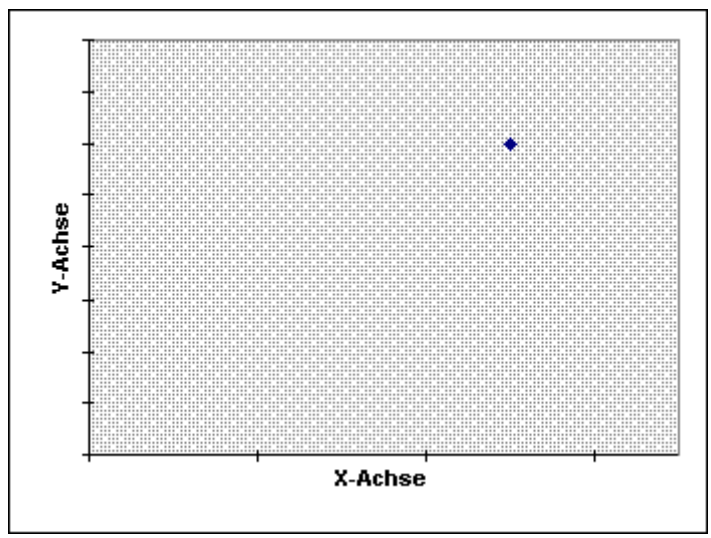

Abbildung 3.9: Ein Punkt in einem Koordinatensystem

Im folgenden wird der Datentyp *Koordinate* für ein zweidimensionales Koordinatensystem definiert.

```
Type Koordinate
   x As Integer
   y As Integer
End Type
```

Auf die einzelnen Komponenten des selbstdefinierten Datentyps greift man über den Datentypnamen und die Datentypkomponenten zu, die durch einen Punkt voneinander getrennt sind.

```
Dim Punkt As Koordinate

Punkt.x = 2
Punkt.y = 3
```

Tabelle 3.7 enthält weitere mögliche Datentypen.

Datentyp	Enthält folgende Felder
Name	Vorname As String Nachname As String
Adresse	Kunde As Name Strasse As String Ort As String
Adressenliste	Ort As String Adressen() As Adresse
Komplex	Real As Double Imaginaer As Double

Tabelle 3.7: Weitere mögliche Datentypen

3.2.6 Schleifen

For-Schleife

Die `For`-Schleife ist eine sogenannte Zählschleife. Ein Startwert, ein Zielwert und die Schrittweite sind vorgegeben. Dabei kann die Schrittweite je nach Bedarf positiv oder negativ sein. Es können auch mehrere Schleifen ineinander verschachtelt werden, um zum Beispiel mehrdimensionale Felder zu berechnen. Das folgende Beispiel berechnet die Summe aller ganzen Zahlen von 0 bis 100.

```
Ergebnis = 0
For i=0 To 100
   Ergebnis = Ergebnis + i
Next i
```

Um die Summe aller ganzen geraden Zahlen zu berechnen, muß die im obigen Beispiel verwendete Schleife um den Parameter `Schrittweite` erweitert werden, da von Null ab nur jede zweite Zahl gerade ist. Man erhält folgenden Programmcode:

```
Ergebnis = 0
For i=0 To 100 Step 2
   Ergebnis = Ergebnis + i
Next i
```

Mit `Exit For` kann eine Schleife vorzeitig abgebrochen werden. Die entsprechende Syntax lautet:

```
For Variable = Startwert To Endwert [Step Schrittweite]
   ...
   Exit ForSchleife vorzeitig beenden
   ...
Next Variable
```

While-Schleife

Die `While`-Schleife ist eine Boolsche Schleife, d.h. solange der Ausdruck wahr ist, wird sie auch durchlaufen. Dies birgt die Gefahr, daß diese Schleife nicht endet, wenn der Boolsche Ausdruck niemals „Falsch“ werden kann. Im Gegensatz zur `For`-Schleife wird die Anzahl der Schleifendurchläufe nicht automatisch mitgezählt. Das folgende Beispiel rechnet eine ganze Zahl in ein Bitmuster um, indem der Rest der Division in eine Zeichenkette gespeichert wird.

```
Zahl = 7
Ergebnis = "0"
While Zahl<1
   Ergebnis = (Zahl Mod 2) & Ergebnis
   Zahl = Zahl / 2
Wend
```

Syntax:

```
While Bedingung
   ...
Wend
```

Do-Schleife

Die `Do`-Schleife ist wie die `While`-Schleife ebenfalls eine Boolsche Schleife, jedoch wird der Programmcode im Gegensatz zur `While`-Schleife immer mindestens einmal durchlaufen, da erst am Ende der

Schleife eine Überprüfung auf weitere Durchläufe erfolgt. Die einfachste Form der Do-Schleife ist eine Endlosschleife.

```
Do
Loop
```

Die Syntax der Do-Schleife ist wie folgt aufgebaut:

```
Do [While/Until Bedingung]
   ...
   Exit DoSchleife vorzeitig verlassen
   ...
Loop
```

oder:

```
Do
   ...
   Exit Do Schleife vorzeitig verlassen
   ...
Loop [While/Until Bedingung]
```

3.2.7 Verzweigungen

Verzweigungen werden gebraucht, um im Programm verschiedene Fälle zu unterscheiden. Bei Wahr/Falsch- oder Ja/Nein-Entscheidungen wird im allgemeinen die If-Then-Else-Verzweigung gewählt. Bei mehreren Auswahlmöglichkeiten greift man auf den Typus der Select Case-Entscheidung zurück.

IF-THEN-ELSE

Die Verzweigung If-Then-Else ist eine sogenannte scharfe Entscheidung, da sie nur zwischen zwei Zuständen unterscheiden kann: Entweder ist der Ausdruck wahr, oder er ist falsch. Die Syntax einer solchen Verzweigung lautet:

```
If Bedingung Then Kommando
If Bedingung Then Kommando Else Kommando
If Bedingung Then
   ...
   Elseif Bedingung Then
   ...
   Else
   ...
End If
```

Diese Art von Entscheidungsmöglichkeit wird benötigt, um Fälle zu unterscheiden, die nur zwei Zustände kennen (Tabelle 3.8). Soll z.B. eine Zahl durch eine Variable dividiert werden, so muß zuvor geprüft werden, ob die Variable nicht 0 ist, da bei einer Division nicht durch 0 dividiert werden darf.

Nachfolgend sind einige prägnante Beispiele zu Ja/Nein-Entscheidungen aufgeführt:

Boolscher Ausdruck	Der Ausdruck ist wahr, wenn...
`X > 0`	...X größer als 0 ist.
`X <> 0`	...X nicht 0 ist.
`Not X`	...X nicht wahr ist.
`X < 0 AND X > 0`	Kann nie wahr werden, da X nicht kleiner als 0 und gleichzeitig größer als 0 sein kann

Tabelle 3.8: Beispiel Boolscher Ausdrücke in If-Then-Else-Abfragen

SELECT CASE

Die Verzweigung `Select Case` ist eine Fallunterscheidung, da sie für mehrere verschiedene Fälle auch unterschiedliche Entscheidungsspielräume zur Verfügung stellt.

```
Select Case Ausdruck
   Case Fall1:
        ...
   Case Fall2:
        ...
   Case Falln:
        ...
   [Case ELSE]
        ...
End Select
```

Diese Art von Verzweigung wird u.a. verwendet, wenn das Programm abhängig vom Wochentag unterschiedliche Programmteile ausführen muß.

Das folgende Beispiel schreibt in eine Variable vom Typ String den Namen des Wochentages, wobei die Woche bei Sonntag beginnt, und Sonntag dem Wert 1 entspricht.

```
Dim Tag As Integer
Dim Wochentag As String
'Der Variablen Tag eine Zahl zwischen
'1 und 7 zuweisen
```

```
Tag = 1
Select Case Tag
    Case 1:
           Wochentag = "Sonntag"
    Case 2:
           Wochentag = "Montag"
    Case 3:
           Wochentag = "Dienstag"
    Case 4:
           Wochentag = "Mittwoch"
    Case 5:
           Wochentag = "Donnerstag"
    Case 6:
           Wochentag = "Freitag"
    Case 7:
           Wochentag = "Samstag"
   Case ELSE
           Wochentag = "Falsche Eingabe"
End Select
```

3.2.8 Funktionen und Prozeduren

Prozeduren (Sub – End Sub)

Prozeduren haben stets den folgenden Aufbau:

```
Sub Prozedurname (Param1 As Vartyp, _
                  ..., Paramn As Vartyp)
   ...
End Sub
```

Als Beispiel wählen wir eine Prozedur, die die Summe zweier Zahlen ermittelt:

```
Sub Summe(Variable1 As Integer, _
          Variable2 As Integer)
   Dim Ergebnis As Integer
   ...
   Ergebnis = Variable1 +  Variable2
   ...
End Sub
```

Funktionen (Function – End Function)

Funktionen sind Prozeduren, die einen Wert an die Funktion zurückgeben, von der sie aufgerufen wurden. Der Aufbau von Funktionen sieht wie folgt aus:

```
Function Name (Param1 As Vartyp, _
               ..., Paramn As Vartyp) AS Vartyp
   ...
   Name = Wert
   ...
End Function
```

Auch hier wieder ein kurzes Beispiel, nämlich eine Funktion, die die Summe zweier Zahlen ermittelt:

```
Function Summe(Variable1 As Integer, _
               Variable2 As Integer) AS Integer

   Summe= Variable1 +  Variable2

End Function
```

Call-by-Value

Call-by-Value bedeutet, daß der Inhalt der Variablen an die aufzurufende Funktion oder Prozedur übergeben wird. Ändert man also den Inhalt der Variablen in der aufgerufenen Funktion oder Prozedur, hat dies keinen Einfluß auf den Inhalt der Variablen der aufrufenden Funktion.

Soll bei einer Funktion oder Prozedur eine Variable durch die Methode *Call-by-Value* übergeben werden, muß dies explizit bei der Deklaration der Funktion angegeben werden.

```
Sub Procedure(ByVal Variablenname As Variablentyp)
Function Function(ByVal Variablenname As Variablentyp)
```

Das folgende Beispiel berechnet die Fakultät einer Zahl. Damit der Variableninhalt in der aufrufenden Funktion nicht verändert wird, wird das Verfahren *Call-by-Value* gewählt.

```
Function Fakultaet(ByVal x As Long) As Long
   Dim i As Integer
   If x > 0 Then
      For i = x - 1 To 2 Step -1
         x = x * i
      Next i
```

```
      Fakultaet = x
   Else
      Fakultaet = 0
   End If
End Function
```

Call-by-Reference

Bei der Methode *Call-by-Reference* wird – im Gegensatz zur Methode *Call-by-Value* – nicht der Inhalt der Variablen übergeben, sondern deren Speicheradresse. Wird der Inhalt der Variablen in der aufgerufenen Funktion oder Prozedur geändert, so ändert sich auch der Inhalt der Variablen in der aufrufenden Funktion. Durch diese Art der Parameterübergabe ist es nun möglich, daß eine Funktion mehrere verschiedene Variablen an die aufrufende Funktion zurückliefert (sog. *indirekte Parameterrückgabe*).

Call-by-Reference muß nicht explizit definiert werden, da diese Art der Variablenübergabe in Visual Basic voreingestellt ist.

Das folgende Beispiel berechnet die Fakultät einer Zahl (vgl. mit Call-by-Value). Diesmal wird die Parameterübergabe jedoch nach dem Prinzip *Call-by-Reference* programmiert.

```
Function Fakultaet(x As Long) As Long
   Dim i As Integer
   If x > 0 Then
      For i = x - 1 To 2 Step -1
         x = x * i
      Next i
      Fakultaet = x
   Else
      Fakultaet = 0
   End If
End Function
```

In dem Beispiel *Call-By-Reference* ist der Inhalt der Variablen der aufrufenden Funktion verändert worden. Er entspricht nicht mehr dem Wert, der der Berechnung zugrundeliegt, sondern dem Ergebnis. In Tabelle 3.9 werden die Variableninhalte der beiden Verfahren nochmals anhand der einzelnen Programmschritte verdeutlicht.

	Call-by-Value	Call-by-Reference
Schritt 1: Initialisierung	`Zahl = 4`	`Zahl = 4`
Schritt 2: Aufruf der Fakultätsfunktion	`Erg = Fakultaet(Zahl)`	`Erg = Fakultaet(Zahl)`
Variableninhalte nach dem Zurückkehren aus der Fakultätsfunktion	`Erg = 24` `Zahl = 4`	`Erg = 24` `Zahl = 24`

Tabelle 3.9: Schrittweiser Vergleich von Call-by-Value und Call-by-Reference

Wird der Inhalt der Variablen in der aufgerufenen Funktion nicht geändert, so ist es nicht relevant, ob die Parameterübergabe mit der Methode *Call-by-Value* oder *Call-by-Reference* erfolgt.

Wird der Inhalt der Variablen jedoch in der aufgerufenen Funktion geändert, so muß bereits bei der Deklaration bekannt sein, ob die aufgerufene Funktion den neuen Wert verwerfen (*Call-by-Value*) oder an die aufrufende Funktion zurückliefern soll (*Call-by-Reference*).

Aufgabe

Das erste Programmbeispiel, das im folgenden entwickelt werden soll, ist eine mathematische Spielerei. Alle natürlichen Zahlen[2] lassen sich in die Zahl 123 umrechnen, indem man sowohl die geraden und ungeraden Ziffern als auch die Anzahl der Stellen ermittelt. Wird dieser Algorithmus oft genug ausgeführt, so lautet das Endergebnis immer 123.

Theoretisches Beispiel mit den Zahlen 3 und 12345

	Zahl	gerade Ziffern	ungerade Ziffern	Anzahl Stellen
Beispiel 1	3	0	1	1
	11	0	2	2
	22	2	0	2
	202	3	0	3
	303	1	2	3
	123	1	2	3
	123	1	2	3
Beispiel 2	12345	2	3	5
	235	1	2	3
	123	1	2	3
	123	1	2	3

Tabelle 3.10: Beispiele zum Algorithmus des Zahlenbeispiels „123“

2. Zahlen, die größer als 0 sind und keine Nachkommastellen haben, also nicht gebrochen sind.

3.3 Das erste Programm

Auf dem leeren Formularfenster werden die Steuerelemente, über die Programm und Benutzer kommunizieren, untergebracht. Hier werden die Buttons, Labelfelder usw. positioniert und der Code für das Formular implementiert. Jedes Formularfenster kann wiederum ein anderes Formularfenster aufrufen. Im Eigenschaftsfenster kann nun der Name des Fensters geänder werden. Hiefür ändern Sie die Eigenschaft `Name` in *Zahlenloch*.

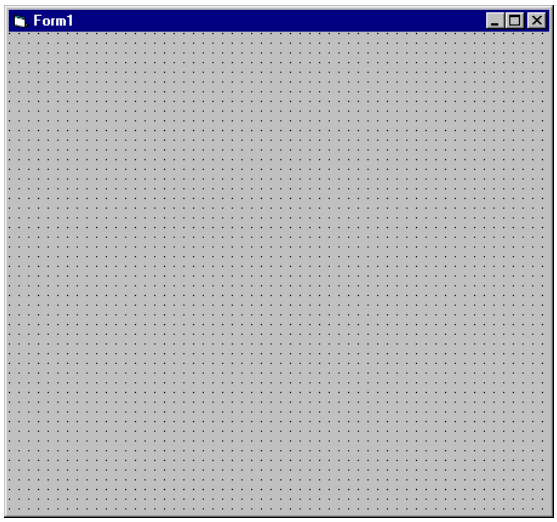

Abbildung 3.10: Ein leeres Formularfenster

Entwickeln einer ersten Applikation mit Visual Basic 6.0

Wenn man mit der linken Maustaste auf dem Formular einen Doppelklick ausführt, gelangt man in den Programmeditor von Visual Basic. Jede Funktion oder Prozedur kann nun wiederum eine andere Funktion oder Prozedur aufrufen.

Um den Programmcode dem aktuellen Fenster zuzuordnen, ist es am einfachsten, wenn mit der linken Maustaste ein Doppelklick auf die *Form* ausgeführt wird. Nach dem Doppelklick wird das Programmeditorfenster geöffnet und der Cursor in der Funktion *Sub Form_Load()* positioniert (Abbildung 3.11). Hier kann nun z.B. dem Fenster ein neuer Titelleistentext zugewiesen werden.

```
Private Sub Form_Load()
   Zahlenloch.Caption = "Zahlenloch 123    Version 1.0"
End Sub
```

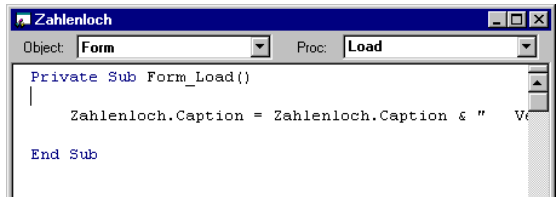

Abbildung 3.11: Programmeditor nach dem Doppelklick auf das Formularfenster

Wird das Programm nun nochmals gestartet, erscheint der eingegebene Satz in der Titelleiste des Fensters. Des weiteren hat das Fenster Eigenschaften wie z.B. einen Namen, Aussehen usw. Diese Eigenschaften lassen sich im Eigenschaftenfenster einstellen. In dem Beispiel „Zahlenloch 123“ erhält das Fenster den Namen „Zahlenloch“, d.h. in dem *Property-Fenster* wird bei der Eigenschaft *Name* der Name „Zahlenloch“ eingetragen. Die Eigenschaft *Caption* wird beim Programmstart in der Funktion *Form_Load* initialisiert, sie braucht also nicht geändert zu werden. Um zu zeigen, daß diese Eigenschaft keinen Einfluß auf das Programmfenster hat, kann der voreingestellte Eintrag *FORM1* gelöscht werden.

3.3.1 Eingabe

Die Dateneingabe in das Programm Zahlenloch

Für das Beispiel wird eine Eingabemöglichkeit benötigt, um die zu berechnende Zahl frei wählen zu können. Zur Eingabe von Daten über die Tastatur steht unter Visual Basic das Texteingabeelement *TextBox* zur Verfügung.

Abbildung 3.12: Das Texteingabeelement „TextBox“

Um eine TextBox oder einen Schaltknopf auf die Oberfläche eines Formulars zu bringen, muß in dem Werkzeugfenster das jeweilige Steuerelement aktiviert werden. Nun muß der Mauszeiger an die Stelle bewegt werden, an der das Element positioniert werden soll. Durch Klicken mit der rechten Maustaste wird die Position des Texteingabeelements auf dem Formular festgelegt. Die Größe des Steuerelements kann durch Ziehen mit gedrückter linker Maustaste definiert werden. Um die Eigenschaften nun für das Beispielprogramm *Zahlenloch* einzustellen, muß das Steuerelement durch einmaliges Anklicken aktiviert werden. Die Eigenschaft *Name* erhält den Wert

Zahl, bei der Eigenschaft *Tag* wird die Zeichenkette „Die zu prüfende Zahl eingeben“ eingegeben. Der Inhalt in der Eigenschaft *Text* wird gelöscht. Bei einem Programmstart kann nun in der TextBox eine Eingabe vorgenommen werden.

3.3.2 Ausgabe

Ausgabe des Programmergebnisses

Natürlich wird auch ein Steuerelement benötigt, mit dessen Hilfe die ermittelten Daten dem Benutzer dargestellt werden. Es gibt dazu mehrere Möglichkeiten, von denen man die jeweils beste wählen kann. Anhand des Beispiels zur Berechnung der Zahl 123 sollen drei Ausgabemöglichkeiten verglichen werden, um das Vorgehen zur Entscheidung zwischen verschiedenen Steuerelementen zu verdeutlichen.

	Label	TextBox	ListBox
Ausgabe von Text möglich	Ja	Ja	Ja
Eingabe von Text während des Programmablaufs möglich	Nein	Ja	Nein
Ausgabe von mehreren Zeilen Text möglich (Möglichkeit zum Verschieben des sichtbaren Textes mittels Bildlaufleiste)	Nein	Nein	Ja

Tabelle 3.11: Vergleich von drei Ausgabe-Steuerelementen

Wie Tabelle 3.11 zeigt, ist es nicht eindeutig, wie viele Schritte notwendig sind, um das Ziel zu erreichen. Es ist aber dennoch sinnvoll, alle Zwischenergebnisse dem Benutzer auszugeben, damit er die Berechnung nachvollziehen kann. Aus der Vergleichstabelle (Tabelle 3.11) ist ersichtlich, daß nur die *ListBox* mehrere Datenzeilen ausgeben kann. Daher wird in diesem Beispiel die *ListBox* eingesetzt.

Um eine *ListBox* auf die Oberfläche eines Formulars zu bringen, muß in der Toolbox das Steuerelement *ListBox* aktiviert werden. Nun muß der Mauszeiger an die Stelle bewegt werden, an der das Element positioniert werden soll. Durch Klicken der rechten Maustaste wird die Position der *ListBox* auf dem Formularfenster festgelegt. Die Größe der *ListBox* kann durch Ziehen mit gedrückter rechter Maustaste definiert werden. Die Eigenschaft *Name* erhält die Definition *Wertigkeit*.

3.3.3 Buttons (Programmsteuerungen)

Ausführen von Programmodulen

Um in einem Programm einen bestimmten Programmteil zu starten, werden *CommandButtons*, kurz *Buttons* oder *Schaltflächen* genannt, eingesetzt. Im Beispiel zur Berechnung der Zahl 123 werden zwei Schaltflächen benötigt – eine zum Starten der Berechnung, eine zum Beenden des Programms.

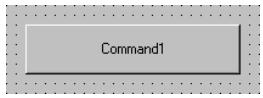

Abbildung 3.13: Die Schaltfläche „Button"

Um einen Button oder Schaltknopf auf die Oberfläche einer Form zu bringen, muß im Werkzeugfenster das Steuerelement *Button* aktiviert werden. Nun muß der Mauszeiger an die Stelle bewegt werden, an der das Element positioniert werden soll. Durch Klicken der rechten Maustaste wird die Position des Buttons auf dem Formular festgelegt. Die Größe des Buttons kann durch Ziehen mit gedrückter rechter Maustaste definiert werden.

Die Eigenschaft *Name* des Buttons erhält den Wert *OK*, die Eigenschaft *Caption* den Wert *Berechnen*. Auf die gleiche Weise muß der Button zum Beenden des Programms auf das Formular aufgebracht werden. Die Eigenschaft *Name* erhält nun den Wert *Abbruch*, und die Eigenschaft *Caption* erhält den Wert *Beenden*.

Im Zusammenhang mit der Schaltflächenprogrammierung kann man gleich auch noch eine nützlichen Funktion einrichten – den sog. *ShortKey*[3]. Damit ein Button mit einem solchen ShortKey aktiviert werden kann, muß in Visual Basic das Zeichen „&" vor denjenigen Buchstaben gesetzt werden, der als ShortKey aktiviert werden soll. Wird zum Beispiel das „&" vor den Buchstaben „d" bei `Beenden` gesetzt, erscheint ein unterstrichenes „d" auf dem Button; das Programm kann nun mit der Tastenkombination Alt-D beendet werden.

Es muß jedoch darauf geachtet werden, daß jeder ShortKey nur einmal benutzt wird, da sonst nicht eindeutig definiert werden kann, welche Funktion als erste ausgeführt wird. Um auch dieses Beispielprogramm an den Windows-Standard anzupassen, erhält der Button OK bei der Eigenschaft *Caption* die Zeichenkette `Be&rechnen` und der

3. Eine Tastenkombination, die das Ausführen einer Menüoption oder einer Funktion ermöglicht, ohne zuvor das Steuerelement aktiviert zu haben.

Button ABBRECHEN – ebenfalls bei der Eigenschaft *Caption* – die Zeichenkette `&Beenden`.

Wenn das Programm neu gestartet wird, können beide Buttons betätigt werden. Dennoch geschieht nichts. Weder wird eine Fehlermeldung ausgegeben, noch wird durch Betätigen des Buttons `Beenden` das Programm abgebrochen. Um das Programm zu beenden, muß immer noch das Tastenkürzel Alt+F4 oder das Visual Basic-Menü AUSFÜHREN | BEENDEN benutzt werden.

Als erstes wird der Button ABBRECHEN programmiert. Durch Doppelklick auf den Button ABBRECHEN wird der Programmeditor mit der Funktion *Abbrechen_Click()* geöffnet. Zwischen dem Funktionsaufruf von *Abbrechen_Click()* und dem Funktionsende `END SUB` kann nun der abzuarbeitende Programmcode eingefügt werden. In diesem konkreten Fall ist das die Funktion *END*. Die Funktion sieht also wie folgt aus:

```
private Sub Abbrechen_Click()
   End
End Sub
```

3.3.4 Der Programmcode zum Algorithmus

Entwickeln des Programmcodes auf der Basis des Vorgabealgorithmus

Der zweite Button soll die eigentliche Berechnungsroutine auslösen. Im folgenden Abschnitt soll diese Funktion programmiert werden. Zum besseren Verständnis wird der Programmablauf zuerst in einem Pseudocode entwickelt und danach mit der jeweiligen Erklärung in einen richtigen Visual Basic-Programmcode umgewandelt.

```
Initialisiere alle Variablen
Solange die Zahl 123 nicht gefunden wurde, durchlaufe diese Schleife
   Zähle alle geraden Zahlen
   Zähle alle ungeraden Zahlen
   Zähle die Anzahl der Ziffern
   Rechne die Ergebnisse in eine neue Zahl um
   Gebe die ermittelte Zahl in der ListBox aus
Ende der Solange-Schleife
```

Die Initialisierung der Variablen

Vorbelegen der Variablen mit den Startwerten

Nachdem der Startbutton aktiviert wurde, muß zuerst geprüft werden, ob es sich bei den eingegebenen Daten um eine erlaubte Zahl handelt. Wenn das zutrifft, wird diese Zahl einer Variablen zugewiesen und der Algorithmus zum Umwandeln der Zahl gestartet.

Der folgende Programmcode führt die Zeile `Initialisiere alle Variablen` aus dem Pseudocode-Listing aus.

```
Dim Ergebnis As String
Dim Gerade As Integer
Dim Ungerade As Integer
Dim i As Integer
If IsNumeric(Zahl.Text) then  'Wenn die Zahl vom Typ
                              'Long ist
   Ergebnis = Zahl.Text
Else
   Exit Sub
End If
Wertigkeit.Clear
```

Die While-Schleife

Ausführen einer Schleife bis zur Erfüllung des Abbruchkriteriums

Nachdem alle Variablen initialisiert wurden, muß die `While`-Schleife realisiert werden. Als Kriterium für die einzusetzende Schleife gilt folgendes:

1. Es muß zu Beginn der Schleife geprüft werden, ob die Zahl 123 schon gefunden wurde, denn wenn 123 als zu modifizierende Zahl eingegeben wird, ist es nicht nötig, den Algorithmus abzuarbeiten.
2. Es ist nicht bekannt, wie oft die Schleife durchlaufen wird. In Tabelle 3.12 ist gut zu erkennen, daß die Größe der Zahl nicht in Zusammenhang mit der Anzahl der Schleifendurchläufe steht, die für das Erreichen der Zahl 123 nötig sind.

Mit einer `While`-Schleife können genau diese Anforderungen erfüllt werden. Der Programmcode in der `While`-Schleife wird solange abgearbeitet, bis das Abbruchkriterium der Schleife wahr ist.

Der Bereich der `While`-Schleife des Pseudocodes stellt sich in Visual Basic wie folgt dar:

```
While Val(Ergebnis) <> 123
   'Zähle alle geraden Zahlen
   'Zähle alle ungeraden Zahlen
   'Zähle die Anzahl der Ziffern
   'Berechne die neue Zahl anhand der Ergebnisse
Wend
```

Zählen der geraden und ungeraden Zahlen

Analysieren aller Ziffern der eingegebenen Zahl

Um Ziffern in einer Zahl zu analysieren, ist es sinnvoll, die Zahl in eine Zeichenkette umzuwandeln und jedes Zeichen (bzw. jede Ziffer) auf das Kriterium „gerade“ bzw. „ungerade“ zu überprüfen. Für die Umwandlung eines Variablentypen in einen anderen existieren folgende Funktionen:

Funktion	Beschreibung
Asc(s)	Liefert den ASCII-Code des ersten Zeichens von *s*.
CBool(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Boolean* um.
CByte(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Byte* um.
CCur(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Currency* um.
CDate(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Date* um.
CDbl(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Double* um.
Chr(n)	Liefert das *n*-te Zeichen des ASCII-Zeichensatzes.
CInt(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Integer* um.
CLng(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Long* um.
CSng(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Single* um.
CStr(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *String* um.
CVar(x)	Wandelt einen beliebigen Datentyp *x* in den Datentyp *Variant* um.
Format(n, "xxx")	Wandelt *n* in eine Zeichenkette unter Berücksichtigung des Formats *xxx* um.
Str(n)	Wandelt *n* in eine Zeichenkette um.
Val(s)	Liefert den Wert der numerischen Zeichenkette *s*.

Tabelle 3.12: Funktionen zur Umwandlung von Datentypen

Im konkreten Fall des Zahlenlochbeispiels wird eine Umwandlungsfunktion von *String* nach *Integer* benötigt, also die Funktion *CInt*. Es ist in diesem Fall nicht relevant, welche der drei Umwandlungsfunktionen eingesetzt wird, da der Eingabewert immer ein Zeichen, also vom Typ *String* ist. Könnte der Eingabewert auch vom Typ *Date* sein, müßte die Funktion *CStr* gewählt werden, da allein die Funktionen *Str* und *Format* numerische Variablen in Zeichenketten umwandeln können.

Um nun die gesamte Zahl Ziffer für Ziffer zu analysieren, muß jede Stelle mit einer Schleife geprüft werden. Hier bietet sich nun eine `For`-Schleife an, da die Anzahl der Schleifendurchläufe bekannt ist. Die Menge der Durchläufe muß mit der Menge der Ziffern, die die Zahl hat, identisch sein. Im Programm ist die Zahl aber als Zeichenkette bekannt, wobei die Anzahl der Zeichen mit der Anzahl der Ziffern der Ausgangszahl identisch ist.

Um nun die Anzahl der Zeichen einer Zeichenkette zu ermitteln, kann die Funktion *Len(s)* eingesetzt werden, die die Anzahl der Zeichen der Zeichenkette *s* zurückliefert.

Der folgende Programmcode ermittelt die Anzahl der geraden und ungeraden Zahlen:

```
Ungerade = 0     'Setze die Variablen zum Zählen der
                 'geraden
Gerade = 0       'und ungeraden Zahlen auf 0

for i = 1 to Len(Ergebnis) 'Solange das Ende der
                           'Zeichenkette nicht erreicht
                           'ist,
   if CInt(Mid(Ergebnis, i, 1)) Mod 2 = 0 Then
                           'prüfe, ob es sich bei der
                           'Ziffer um eine
      Gerade = Gerade + 1 'gerade Zahl
   Else                    'oder
      Ungerade = Ungerade + 1 'um eine ungerade Zahl
                           'handelt
   End If
next i                     'Prüfe die nächste Ziffer
```

Die in diesem Programmcode enthaltene Funktion *Mid$(s, i, l)* dient dazu, einen Teil der Zeichenkette *s* zurückzuliefern. Dabei handelt es sich um den Teil, der sich in der Zeichenkette *s* ab Position *i* befindet und die Länge *l* hat.

Befehl	Ergebnis
`Mid$("Dies ist eine Zeichenkette", 2, 9)`	`ies ist e`
`Mid$("Dies ist eine Zeichenkette", 23, 8)`	`ette`
`Mid$("Dies ist eine Zeichenkette", 3, 0)`	

Tabelle 3.13: Beispiel für die Funktion „Mid$“

Die Funktion Mod der folgenden Befehlszeile

```
if CInt(Mid$(Ergebnis, i, 1)) Mod 2 = 0 Then
```

entspricht der Funktion *Modulo*. Die Funktion *Modulo* liefert immer den Restwert einer Division zurück. Hierzu ein Beispiel:

$\frac{7}{2} = 3$ *Rest 1* *7 Modulo 2 = 1*

$\frac{8}{2} = 4$ *Rest 0* *8 Modulo 2 = 0*

Ermitteln des Ergebnisses und Berechnen der neuen Zahl

Berechnen der neuen Zahl anhand der analysierten Ziffern der Ausgangszahl

An dieser Stelle des Programms sind alle drei Werte, die zur Berechnung der neuen Zahl benötigt werden, bekannt: die Anzahl der ungeraden und geraden Zahlen sowie – mit Hilfe der Funktion *Len* – die Anzahl der Ziffern der Ausgangszahl. Diese Daten bilden nun die neue Zahl. Die einzelnen Zahlen müssen jedoch nicht nach einem bestimmten Algorithmus addiert, sondern wie einzelne Zeichenketten aneinandergereiht werden.

Die folgende Programmzeile führt diese Aneinanderreihung aus:

```
Ergebnis = CLng(CStr(Gerade) & CStr(Ungerade) & CStr(Len(Ergebnis)))
```

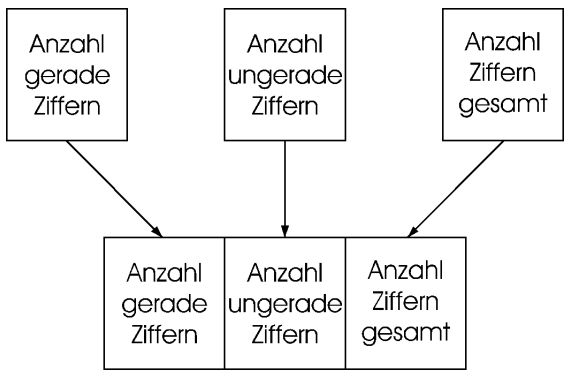

Abbildung 3.14: Verknüpfung der ermittelten Ergebnisse

Ausgabe des Ergebnisses

Zu guter Letzt muß das Ergebnis auch noch auf dem Bildschirm des Benutzers ausgegeben werden. Wenn bei jedem Schleifendurchlauf die ermittelte Zahl in die ListBox geschrieben wird, sind alle Berechnungen über die Zwischenschritte nachvollziehbar. Die folgende Programmzeile schreibt das neu berechnete Ergebnis immer in die erste Zeile der ListBox *Ergebnis*:

```
Wertigkeit.Additem CStr(Ergebnis), 0
```

Die gesamte Funktion zur Berechnung des Zahlenloches 123

Nun sind alle Befehlszeilen des Pseudocodes in Visual Basic-Programmcode umgewandelt worden. Im folgenden ist nun die gesamte Funktion, die zum Button OK gehört, abgebildet.

```
Private Sub Ok_Click()
   Dim Ergebnis As String
   Dim Gerade As Integer
   Dim Ungerade As Integer
   Dim i As Integer
   If IsNumeric(Zahl.Text) then  'Wenn die Zahl
                                 ' vom Typ Long ist
      Ergebnis = Zahl.Text
   Else
      Exit Sub
   End If

   While Val(Ergebnis) <> 123
      Ungerade = 0     'Setze die Variablen zum
                       ' Zählen der geraden
      Gerade = 0       'und ungeraden Zahlen auf 0

      for i = 1 to Len(Ergebnis) 'Solange das Ende
                       'der Zeichenkette nicht
                       'erreicht ist,
         if CInt(Mid(Ergebnis, i, 1)) Mod 2 = 0 Then
                       'prüfe, ob es sich bei der
                       'Ziffer um eine
            Gerade = Gerade + 1 'gerade Zahl
         Else          'oder
            Ungerade = Ungerade + 1 'um eine
                       'ungerade Zahl handelt
         End If
      next i           'Prüfe die nächste Ziffer


      Ergebnis = CLng(CStr(Gerade) & _
                 CStr(Ungerade) & _
                 CStr(Len(Ergebnis)))
      Wertigkeit.Additem CStr(Ergebnis), 0
   Wend
End Sub
```

Wenn das Programm gestartet wird, kann die Oberfläche wie in Abbildung 3.15 dargestellt aussehen.

Abbildung 3.15: Mögliche Oberfläche des Programms „Zahlenloch 123"

Um das Programm zu testen, ist es sinnvoll, Werte zu wählen, bei denen das Ergebnis und die Zwischenschritte bekannt sind. In diesem Fall bieten sich die Zahlen aus der Abbildung 3.15 an, da für diese Zahlen schon alle Zwischenschritte berechnet wurden. In der ListBox müssen alle Zahlen in der gleichen Reihenfolge wie in der Beispieltabelle auftreten.

3.3.5 Größere Benutzerfreundlichkeit

Erhöhen der Benutzerfreundlichkeit durch Einsatz einer Statuszeile

Bei vielen Programmen erhält man in Form einer sog. *Statusleiste* für die meisten Steuerelemente auf der Programmoberfläche eine schnelle Hilfe. In Visual Basic ist diese schnelle Hilfe recht einfach zu realisieren.

Als Statusleiste wird ein Steuerelement benötigt, das der Ausgabe dient, in dem aber keine Eingabe gemacht werden darf. Darüber hinaus ist es nicht möglich, mehrere Datensätze darzustellen. In der Tabelle 4.11 wurden verschiedene Ausgabesteuerelemente verglichen. Aus diesem Grund wird für die Statusleiste das Steuerelement *Label* verwendet.

Wie alle anderen Steuerelemente wird die Statusleiste durch Aktivieren des Symbols im Werkzeugfenster und durch Festlegen der Position und Größe infolge von Anklicken und Ziehen des Steuerelements mit Hilfe der Maus erzeugt. Die Statusleiste läßt sich optisch hervorheben, indem bei der Eigenschaft *BorderStyle* die Option *1-FestEinfach* (Abbildung 3.15) eingestellt wird.

Mit der Statusleiste wird erreicht, daß zu jedem Steuerelement eine Hilfetext ausgegeben werden kann, wenn der Mauszeiger über das jeweilige Steuerelement bewegt wird.

Fast jedes Steuerelement besitzt die Eigenschaft *Tag*, die keinen Einfluß auf das Steuerelement selbst hat. In dieser Eigenschaft kann zu jedem Steuerelement ein kurzer Beschreibungstext eingegeben werden.

Steuerelement	Beschreibungstext
Button Ok	Hiermit wird das Programm gestartet.
Button Abbrechen	Hiermit wird die Berechnung beendet.
ListBox	Hier sind alle Zwischenschritte der Berechnung enthalten.
TextBox	Bitte geben Sie hier die zu berechnende Zahl ein.
Form	

Tabelle 3.14: Beschreibungstexte der Steuerelemente[4]

Der Eigenschaft *Tag* des Formularfensters *Form* ist eine sog. *leere* Zeichenkette zugewiesen worden, da in der Statuszeile auch kein Hilfetext ausgegeben werden soll, wenn der Mauszeiger auf keines der Steuerelemente zeigt. Um den Inhalt der Eigenschaft *Tag* eines jeden Steuerelements der Eigenschaft *Caption* der Statusleiste zuzuweisen, wird die Funktion *MouseMove* benötigt. *MouseMove* wird dann ausgeführt, wenn das jeweilige Steuerelement mit der Maus angesteuert wird. Für jedes Steuerelement, das in der Statusleiste einen Hilfetext enthalten soll, muß die Funktion *MouseMove* wie folgt angepaßt werden:

Form

```
Private Sub Form_MouseMove(Button As Integer, Shift As Integer, X As Single, Y As Single)

   Status.Caption = Form.Tag

End Sub
```

Der Button Ok

```
Private Sub Ok_MouseMove(Button As Integer, Shift As Integer, X As Single,
Y As Single)
   Status.Caption = Ok.Tag
End Sub
```

4. In der Eigenschaft *Tag* der *Form* ist eine Zeichenkette mit der Länge 0 eingetragen

Der Button Abbrechen

```
Private Sub Abbruch_MouseMove(Button As Integer, Shift As Integer, X As
Single, Y As Single)
   Status.Caption = Abbruch.Tag
End Sub
```

Die ListBox zur Ausgabe der Zwischenschritte

```
Private Sub Wertigkeit_MouseMove(Button As Integer, Shift As Integer, X As
Single, Y As Single)
   Status.Caption = Wertigkeit.Tag
End Sub
```

Die TextBox zur Eingabe der Ausgangszahl

```
Private Sub Zahl_MouseMove(Button As Integer, Shift As Integer, X As Sin-
gle, Y As Single)
   Status.Caption = Zahl.Tag
End Sub
```