Jürgen Kotz

erfolgreich Visual Basic 2010 programmieren

ADDISON-WESLEY

3

Visual Basic 10 allgemein

Ein Schwerpunkt dieses Buchs liegt darin, die wichtigsten sprachlichen Merkmale von Visual Basic 10 aufzugreifen und Ihnen näherzubringen, wie man mit diesen umfangreiche Applikationen schreiben kann. Dieses Kapitel behandelt allgemeine Dinge, die für das Programmieren notwendig sind. Visual Basic wird von Kritikern oft als simple Programmiersprache bezeichnet. Doch seit dem .NET Framework ist es ohne Weiteres möglich, auch komplexe Applikationen mit Visual Basic zu entwickeln. Falls Sie noch nicht mit Visual Basic unter .NET vertraut sind oder gerade von Visual Basic 6 umsteigen, kann Ihnen dieses Kapitel die wichtigsten Grundkenntnisse in Visual Basic 10 vermitteln.

3.1 Datentypen

Variablen gehören zu den wichtigsten Bestandteilen einer jeden Programmiersprache. Variablen werden mit einem Datentyp definiert, der festlegt, welche Art von Daten in der Variablen gespeichert werden können. Die allgemeine Syntax lautet:

```
Dim VariablenName As Datentyp
```

Visual Basic 10 achtet ganz genau darauf, ob der Wert, der in die Variable geschrieben wird, auch tatsächlich zum Datentyp passt.

Folgender Code würde eine Compiler-Fehlermeldung hervorrufen, da der Datentyp *Integer* nur für ganze Zahlen gedacht ist und keinen Text speichern kann.

```
Dim Name As Integer
Name = "Max Mustermann"
```

Visual Basic 2010 kennt die in Tabelle 3.1 aufgelisteten Datentypen:

Tabelle 3.1
Datentypen im Überblick

Datentyp	Bedeutung
Boolean	Wahr-/Falsch-Wert (True/False)
Byte	Ganzzahl zwischen 0 und 255
Char	Unicode-Zeichen
Date	Datumswert vom 1.1.0001 bis zum 31.12.9999
Decimal	Eine Ganzzahl mit 128 Bit mit maximal 28 Stellen nach dem Trennzeichen
Double	Fließkommazahl mit 15 Stellen nach dem Komma
Integer	Ganzzahl mit der Größe von 4 Byte
Long	Ganzzahl mit der Größe von 8 Byte
Object	Basisklasse aller komplexen Typen
SByte	Ganzzahl zwischen -128 und 127
Short	Ganzzahl mit der Größe von 2 Byte
Single	Fließkommazahl
String	Zeichenkette mit bis zu 2 Milliarden Zeichen
UShort	Positive Ganzzahl zwischen 0 und 65535
UInt	Positive Ganzzahl mit 4 Byte
ULong	Positive Ganzzahl mit 8 Byte
BigInteger	Für Ganzzahlen, die außerhalb des Long-Bereichs liegen

Jeder Typ hat einen eigenen Namen (z.B. String) und eine bestimmte Größe. Die Größe gibt an, wie viele Bytes jedes Objekt im Speicher belegt. Jeder Datentyp in Visual Basic 10 gehört zu einem .NET-Datentyp. Was Visual Basic *Integer* nennt, ist in .NET unter *Int32* bekannt. Das ist für Sie von Bedeutung, falls Sie vorhaben, Objekte für mehrere Programmiersprachen im .NET Framework bereitzustellen. Sie können innerhalb von Visual Basic sowohl die VB-Bezeichnung angeben wie auch den zugrunde liegenden .NET-Typen.

3.1.1 Variablen

Eine Variable ist kurz gesagt ein Objekt, das einen bestimmten Wert enthält. Zum Beispiel:

```
Dim text As String = "Hallo Welt"
```

In diesem Beispiel ist `text` ein `String`-Datentyp. Bei der Initialisierung wird der Variablen der Wert `"Hallo Welt"` zugeordnet. Die eigentliche formale Definition, wie eine Variable deklariert wird, lautet:

```
Access-Modifikator Bezeichner As Datentyp (= Wert)
```

Bezeichner ist der Fachausdruck für die Namen von Elementen, die im Code vorkommen können, wie zum Beispiel Klassen, Methoden oder eben Variablen. Hier bezieht sich dieser Begriff auf den Variablennamen. Im weiter oben genannten konkreten Beispiel ist der Bezeichner `text`.

Info

`Dim` ist die Abkürzung für Dimension. Diese Abkürzung existiert beispielsweise auch in der Mathematik und in der Physik. Sie lässt sich auf die alten Tage der BASIC-Programmierung zurückführen.

Es ist nicht immer notwendig, bei der Deklaration einer Variablen einen Wert zuzuweisen. Falls Sie das nicht tun, wird Visual Basic 10 in Abhängigkeit vom Datentyp Standardwerte vergeben. Diese können Sie der Tabelle 3.2 entnehmen:

Tabelle 3.2
Initialwerte von Datentypen

Datentyp	Wert
Boolean	False
Date	01/01/0001 12:00:00 AM
Decimal	0
Object	Nothing
String	""
Alle numerischen Typen	0

Nachdem nun einige Datentypen vorgestellt wurden, will ich noch kurz darauf eingehen, wie diese Typen im Speicher repräsentiert werden.

Prinzipiell gibt es zwei unterschiedliche Arten von Datentypen: Wertetypen und Referenztypen.

Bei den Wertetypen werden die Daten direkt im Speicher abgebildet, wie es bei den bislang vorgestellten Datentypen, mit Ausnahme von Strings, der Fall ist. Hinter der Variablen steht also direkt deren Wert.

Die zweite Art, Daten zu repräsentieren, erfolgt über Referenzen auf die eigentlichen Daten. Dies ist bei komplexen Datentypen, bei denen die tatsächliche Größe variabel oder unbestimmt ist, der Fall. Zu diesen Datentypen gehören Strings, aber auch Arrays und die benutzerdefinierten Datentypen. Man spricht deshalb von Referenztypen.

3.1.2 Wertetyp- und Referenztypsemantik

Anwendungen nutzen im RAM prinzipiell zwei Bereiche, den Stack und den Heap. Nur die Variablen im Stack besitzen Namen, Objekte im Heap sind namenlos. Dort kann man nur über Adressen auf Daten zugreifen.

Wie gerade erwähnt, unterscheiden sich Variablen in zwei Gruppen: Wertetypen und Referenztypen.

Bei den Wertetypen wird der Wert der Variablen direkt an der Speicherstelle im Stack gespeichert.

Bei den Referenztypen wird an der Speicherstelle im Stack auf eine Adresse im Heap referenziert. Sie werden somit als Zeiger auf einen anderen Speicherbereich implementiert.

Doch Wertetypen und Referenztypen unterscheiden sich nicht nur in der Art der Speicherung, sondern auch in der Art und Weise, wie Daten zugewiesen werden.

Wertetypsemantik

Bei der Zuweisung eines Wertetyps wird eine Kopie der entsprechenden Variablen angefertigt. Wird die zugewiesene Variable manipuliert, hat das keine Auswirkungen auf die Originalvariable.

```
Dim zahl1 As Integer = 7
Dim zahl2 As Integer = zahl1
zahl2 = 9
```

In diesem Beispiel wird eine Variable `zahl1` vom Typ `Integer` definiert und dabei der Wert 7 zugewiesen. Im nächsten Schritt wird eine zweite Integervariable `zahl2` definiert und ihr wird der Wert von `zahl1` zugewiesen. Der Wert von `zahl1` wird aufgrund der Wertetypsemantik in die Speicherzelle `zahl2` kopiert. Anschließend wird der Wert von `zahl2` auf 9 gesetzt. Wenn beide Variablen danach ausgegeben werden, dann haben sie auch unterschiedliche Werte. Einmal den Wert 7 für `zahl1` und 9 für `zahl2`. Sie sehen, dass bei der Zuweisung `zahl2 = zahl1` eine Kopie des Werts durchgeführt wurde.

Referenztypsemantik

Bei der Zuweisung eines Referenztyps hingegen wird der Wert der Variablen nicht kopiert. Vielmehr wird die Adresse im Heap, die auf das tatsächliche Objekt zeigt, kopiert. Somit zeigen beide Variablen auf denselben Speicherbereich. Wird die zugewiesene Variable manipuliert, hat das Auswirkungen auf die Originalvariable.

```
Dim p1 As New Person()
p1.Name = "Julia"
Dim p2 As Person = p1
p2.Name = "Lennard"
```

In diesem Beispiel wird eine Variabel `p1` vom Typ `Person` (ohne der Objektorientierung vorgreifen zu wollen, sehen Sie die Definition dieses Typs in Listing 3.1) definiert und instanziert. Dann wird der Eigenschaft `Name` der Wert `"Julia"` zugewiesen. Im nächsten Schritt wird eine zweite Personenvariable `p2` definiert und ihr wird `p1` zugewiesen. Dabei wird die Adresse von `p1` aufgrund der Referenztypsemantik in die Speicherzelle `p2` kopiert. Anschließend wird die Name-Eigenschaft von `p2` auf `"Lennard"` gesetzt. Wenn von beiden Variablen danach die Name-Eigenschaft ausgegeben wird, dann haben sie beide identische Werte, nämlich `"Lennard"`. Bei der Zuweisung `p2 = p1` wurden also nicht etwa die eigentlichen Daten des Person-Objekts, sondern allein die Adresse kopiert.

```
Public Class Person
    Private _name As String
    Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal value As String)
            _name = value
        End Set
    End Property
End Class
```

Listing 3.1 Definition einer Klasse Person

Achtung

Obwohl es sich beim Datentyp `String` um einen Referenztyp handelt, verhält er sich wie ein Wertetyp. `String` ist somit ein Referenztyp mit Wertetypsemantik. Die Ausnahme wurde von den Framework-Entwicklern gemacht, um die Verarbeitung dieses doch wesentlichen Datentyps intuitiver zu implementieren.

3.1.3 Nullable Typen

Die vorher gerade vorgestellten Wertetypen werden bei der Definition mit einem Standardwert versehen, sofern Sie nicht direkt einen Initialwert zuweisen (siehe Tabelle 3.2).

Wertetypen besitzen somit immer einen Wert, auch wenn es eben nur der Initialwert ist. Ein Wertetyp kann niemals den Wert `Nothing` annehmen. Doch gerade bei Datenbankanwendungen ist es sinnvoll, Spalten, die den Wert `null` (nicht belegt) besitzen dürfen, nicht mit einem Initialwert zu belegen. Stellen Sie sich vor, in einer Mitarbeitertabelle gibt es für einen Mitarbeiter ein Feld *AnzahlKinder*. Wenn der Wert nicht bekannt ist, wird eben nichts eingetragen, also `null`, und somit weiß jeder, dass diese Information nicht bekannt ist. Wird jedoch der Initialwert eingetragen, würde man daraus schließen, dass der Mitarbeiter keine Kinder hat, was unter Umständen eine falsche Information darstellt.

Mit dem .NET Framework 2.0 wurden sogenannte *Nullable Types* eingeführt. Das bedeutet, dass Wertetypen jeweils ein Nullable-Äquivalent besitzen, das auch den Wert `Nothing` annehmen kann.

Einen **Nullable Integer** definieren Sie dabei wie folgt:

```
Dim Arg1 As Nullable(Of Integer)
```

NEW

Seit der Einführung der Vorgängerversion Visual Basic 9 können Sie die Definition jetzt auch wie folgt angeben:

```
Dim Arg1 as Integer?
```

3.1.4 Konstanten

Konstanten beinhalten beliebige Werte, die sich, im Gegensatz zu Variablen, über die gesamte Laufzeit des Programms nicht ändern. Der Vorteil von Konstanten ist, dass bei festen Werten eine Änderung meist durch Austauschen und Anpassung einer Zeile möglich ist. Falls Sie zum Beispiel an einem Programm arbeiten, das die Besoldung von Mitarbeitern nach einem festen Stundenlohn ausrechnet, wäre es ratsam, diesen festen Stundenlohn als Konstante festzulegen. Andernfalls müssten Sie wahrscheinlich mehrmals im Code den entsprechenden Stundenlohn als Zahl eingeben. Bei Änderung des Stundenlohns würde im zweiten Fall eine aufwändige Änderung des Codes nötig sein, während Sie bei einer Konstante nur einen Wert ändern müssten.

Beispiele für die Definition von Konstanten sehen Sie in Listing 3.2:

Listing 3.2 Festlegung von Konstanten

```
Const STUNDENLOHN As Integer = 23
Const ARBEITSSTUNDEN As Integer = 8
Const LOHNPROTAG As Integer = _
    STUNDENLOHN * ARBEITSSTUNDEN
Const WindowsPfad As String = "C:\Windows\"
```

Konstanten können, wie Variablen, direkt im Programm mit ihrem Namen angesprochen werden. Der Wert einer Konstanten kann nach der Definition nicht mehr verändert werden. Ein entsprechender Versuch würde vom Compiler mit einer Fehlermeldung beantwortet.

3.1.5 Aufzählungen (Enums)

Enums helfen Ihnen, konstante Werte einer Gruppe zuzuordnen. Nehmen wir an, Sie entwickeln eine Applikation für die Verwaltung von Mitarbeitern. Hierzu teilen Sie die Mitarbeiter in verschiedene Kategorien wie Fertigung, Einkauf, Verkauf, Marketing etc. ein. Sie können nun eine Enum als eine Art Liste anlegen.

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf
    Verkauf
    Marketing
End Enum
```

Falls von Ihnen nicht anders festgelegt, ordnet das .NET Framework nun jedem der Auflistungswerte einen numerischen Wert zu, angefangen bei 0 in aufsteigender Reihenfolge. Alternativ können Sie im Code auch direkt die Enum-Konstanten verwenden, natürlich mit IntelliSense-Unterstützung. Das hat unter anderem den Vorteil, dass Sie sich die Bedeutung der Indizes nicht merken müssen, und es erhöht auch die Lesbarkeit des Quellcodes.

Das folgende Beispiel (Listing 3.3) soll die Funktionsweise von Enums verdeutlichen:

Listing 3.3
Verwendung einer Enum

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf
    Verkauf
    Marketing
End Enum

Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = KatMitarbeiter.Verkauf
    MessageBox.Show(mitarbeiter)
End Sub
```

Bei Ausführung des Programms wird die MessageBox den Wert 2 ausgeben, da der Name `Verkauf` in der Enum den Wert 2 enthält.

Falls Sie ermitteln wollen, welches Element sich hinter einem Wert in der Aufzählung verbirgt, lässt sich das recht simpel bewerkstelligen, indem Sie an der Enumerationsvariablen die `ToString()`-Methode aufrufen:

```
Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = KatMitarbeiter.Verkauf
    MessageBox.Show(mitarbeiter.ToString)
End Sub
```

Solange nicht anders festgelegt, werden die Elemente als Integer angelegt. Falls gefordert, können Sie auch einen anderen Datentyp für die Aufzählungselemente festlegen. Eine Definition als Byte, Short oder Long reicht vollkommen aus.

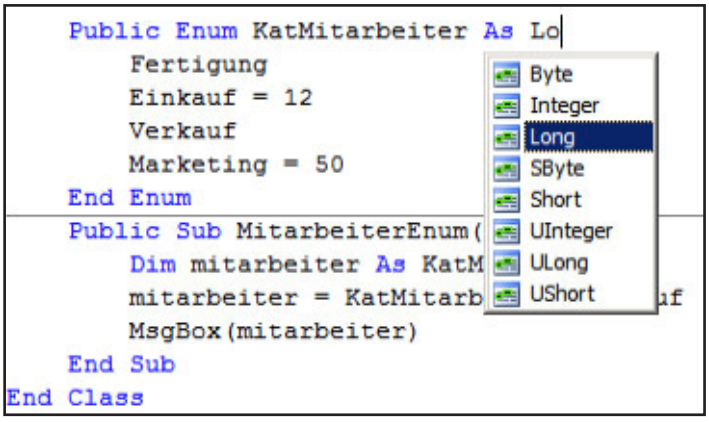

Abbildung 3.1
Definieren des Datentyps in einer Enum

Falls Sie, aus welchen Gründen auch immer, nicht mit der Zählung der Enums zufrieden sind, können Sie auch selbst festlegen, welche Nummer einem Enumerationswert zugeordnet werden soll.

```
Public Enum KatMitarbeiter
    Fertigung
    Einkauf = 12
    Verkauf
    Marketing = 50
End Enum
```

In diesem Beispiel würde die Ausgabe der Werte wie in Tabelle 3.3 aufgelistet lauten:

Tabelle 3.3 Enum-Übersicht

Enum	Wert
Fertigung	0
Einkauf	12
Verkauf	13
Marketing	50

Was aber nun, wenn einige Mitarbeiter verschiedene Aufgaben im Betrieb haben, beispielsweise im Einkauf und Verkauf gleichzeitig beschäftigt sind? Für diesen Fall liefert Visual Basic natürlich auch eine Lösung, sogenannte `Flags`-Enums oder zu Deutsch Flags-Aufzählungen.

Sie sollten hierbei jedem Eintrag einen Wert zuweisen, der sich bitweise kombinieren lässt. Ebenfalls sollten Sie einen neuen Eintrag mit dem Namen None anlegen, der den Wert 0 enthält.

```
<Flags()> Public Enum KatMitarbeiter
    Fertigung = 1
    Einkauf = 2
    Verkauf = 16
    Marketing = 32
    None = 0
End Enum
```

Nun kann eine Enumerationsvariable gleichzeitig mehrere Enum-Werte halten. Diese können Sie zum Beispiel mit dem bitweisen Or-Operator abfragen. Der Code in Listing 3.4 soll das illustrieren:

Listing 3.4 Abfrage von Enum-Werten mit Or

```
Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatMitarbeiter
    mitarbeiter = _
KatMitarbeiter.Verkauf Or KatMitarbeiter.Marketing
    If mitarbeiter = KatMitarbeiter.Verkauf Or
        KatMitarbeiter.Marketing Then

        MessageBox.Show(mitarbeiter.ToString)
        'Ausgabe: Verkauf, Marketing
    End If
End Sub
```

3.1.6 Konvertierung in verschiedene Datentypen

Visual Basic 10 bietet die Möglichkeit, Objekte in verschiedene Datentypen zu konvertieren. Nehmen wir an, Sie haben eine *String*-Variable mit dem Inhalt `"24"` und möchten damit eine Berechnung durchführen. Wie Sie sicherlich wissen, ist es zwar möglich, mit Zeichenketten zu rechnen (diese werden vor Verwendung in den Typ `Integer` konvertiert), jedoch würde die `Option Strict On`-Eigenschaft das nicht erlauben (zu `Option Strict On` erfahren Sie im späteren Teil dieses Kapitels mehr). Daher wäre es ratsam, den Wert der Variablen vor der Berechnung nach `Integer` zu konvertieren. Das funktioniert so, wie Sie es in Listing 3.5 sehen:

Listing 3.5
Konvertierung in verschiedene Datentypen

```
Dim strZahl As String
Dim intZahl As Integer
' Zahl vom Typ String
strZahl = "24"
' Konvertierung in einen Integer-Wert
intZahl = Convert.ToInt32(strZahl)
'Rechenoperation mit dem neuen Integer-Wert
intZahl = intZahl * 100
' Ausgabe des Wertes in einer Messagebox
' mit voriger
' Konvertierung in einen String-Wert
MessageBox.Show(intZahl.ToString)
```

Nach der Konvertierung in einen Integer-Wert kann mit diesem gerechnet werden. Um das Ergebnis wieder typengerecht auszugeben, erfolgt die Konvertierung zurück in einen `String`.

In der Tabelle 3.4 finden Sie die jeweiligen Methoden der `Convert`-Klasse, um Datentypen zu konvertieren:

Tabelle 3.4
Konvertierungsmöglichkeiten

Methode	Konvertierung in folgendes Format
Convert.ToString()	String
Convert.ToSingle()	Single
Convert.ToDouble()	Double
Convert.ToBoolean()	Boolean
Convert.ToByte()	Byte
Convert.ToSByte()	SByte
Convert.ToChar()	Char
Convert.ToInt16()	Short
Convert.ToInt32()	Integer
Convert.ToInt64()	Long
Convert.ToUInt16()	UShort
Convert.ToUInt32()	UInteger
Convert.ToUInt64()	ULong
Convert.ToDateTime()	Date
Convert.ToDecimal()	Decimal

Bei einer Umwandlung von einem Datentyp in einen kompatiblen größeren Datentyp (zum Beispiel von `Integer` nach `Long`) brauchen Sie keinen expliziten Code zu schreiben, diese Typumwandlung führt der Compiler automatisch durch. Man spricht in diesem Zusammenhang von einer impliziten Typumwandlung.

3.1.7 Die CType-Methode

Eine weitere Möglichkeit zur Umwandlung von Datentypen bietet Ihnen in Visual Basic 10 die `CType()`-Methode.

Die Syntax der `CType()`-Methode sieht wie folgt aus:

```
CType(Variable, Datentyp)
```

Dabei ist `Variable` ein Platzhalter für die Variable, die den entsprechenden Wert enthält, und `Datentyp` ist der Datentyp, in den die Variable umgewandelt werden soll.

Das folgende Beispiel wandelt eine Variable vom Typ `Object` in ein Objekt vom Typ `Button` um.

```
Private Sub Button1_Click( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles Button1.Click
    Dim b As Button = CType(sender, Button)
End Sub
```

Eine Alternative zu `CType()` ist die `DirectCast()`-Methode, die sehr ähnlich funktioniert.

3.1.8 GetType-Methode

Da es Polymorphie und Vererbung – jeder Datentyp ist von Object abgeleitet – erlauben, dass ein Objekt einen allgemeineren Datentyp besitzen kann als den, mit dem es ursprünglich definiert wurde, ist es oft wichtig, festzustellen, welchen Datentyp eine Variable zur Zeit der Ausführung besitzt. Um sicherzugehen, welchen Datentyp ein Objekt gerade besitzt, kann dieser mit der `GetType()`-Funktion ermittelt werden. Das folgende Beispiel in Listing 3.6 verdeutlicht die Verwendung von `GetType()`. Zunächst wird eine Variable vom Typ `Object` deklariert. Dieser wird ein Wert zugewiesen, anschließend wird in einer `MessageBox` mit `GetType()` der Typ des Objekts ermittelt. Als Ausgabe würde »Der Typ der Variablen ist System.String« erscheinen.

Listing 3.6 Verwendung von GetType()

```
Dim Text As Object
Text = "Dies ist ein Test"
MessageBox.Show("Der Typ der Variable ist " & _
    Text.GetType.ToString)
```

3.1.9 Option Strict On

Standardmäßig erlaubt Visual Basic 10 die Konvertierung eines Datentyps in einen beliebigen anderen. Das kann zu Datenverlust führen, vor allem dann, wenn die Größe des neuen Datentyps überschritten wird. In Visual Basic 6 war es beispielsweise möglich, Zahlen aus dem Typ String mit Integerzahlen zu addieren. Das ermöglicht zwar eine bequeme und schnelle Programmierung, kann jedoch in größeren Projekten zu schwer auffindbaren Fehlern führen.

In Visual Basic 10 hat sich dieses Problem nicht grundlegend geändert, es gibt jedoch den Modus `Option Strict On`, der implizite Typenumwandlung nicht mehr erlaubt.

Sie sollten `Option Strict On` immer benutzen, um schwerwiegende Folgefehler von vornherein auszuschließen – auch wenn die Entwicklung dadurch länger dauern könnte.

Im Übrigen steht der Visual Basic-Editor dem Programmierer mit konkreten Vorschlägen zur Seite, falls ein Datentyp Gefahr läuft, falsch konvertiert zu werden. Schauen Sie sich das Beispiel in Listing 3.7 einmal näher an:

```
Dim strZahl As String
Dim dblZahl As Double
strZahl = "24"
dblZahl = strZahl * 100
```

Listing 3.7
Unzulässige Rechenoperation mit einem String

Ohne `Option Strict On` gäbe der Editor keine Meldung aus, die Rechenoperation würde ohne weitere Prüfung durchgeführt werden. Bei größeren Zahlen, mit denen noch dazu umfangreiche Berechnungen angestellt würden, ist die Gefahr des Datenverlusts jedoch nicht auszuschließen, was fehlerhafte Ausgaben zur Folge hätte.

Wenn wir nun `Option Strict On` hinzuschalten, was Sie eigentlich immer tun sollten, würden wir einen Compiler-Fehler angezeigt bekommen.

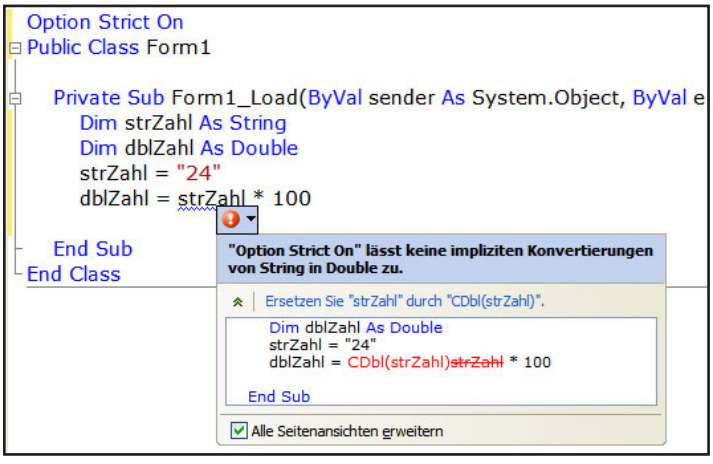

Abbildung 3.2
Meldungsbox bei unzulässiger Verwendung von Variablen bei der Compiler-Einstellung Option Strict On

Wie Sie sehen, unterbreitet Ihnen Visual Studio gleich einen Vorschlag, wie Sie das Problem lösen können. Mit einem Klick wird `strZahl` zu `CDbl(strZahl)` und Sie können mit der Arbeit an Ihrem Programm fortfahren.

Tipp

`Option Strict On` können Sie in Visual Studio 2010 in Ihren Projekteigenschaften festlegen. Für jedes neue Projekt, das Sie beginnen, müssen Sie diese Eigenschaft erneut festlegen oder in der Projektvorlage diese Eigenschaft als Standard für jedes neue Projekt setzen. Sie können ebenfalls an oberster Stelle der Codedatei folgende Anweisung hinzufügen: `Option Strict On`.

3.1.10 Programmiernotation

Gerade für Fortgeschrittene stellt sich oft die Frage, wie Objekte, wie zum Beispiel Textfelder, Group-Boxen, Buttons etc., oder auch Variablen und Arrays richtig benannt werden sollen. Viele Programmierer benennen Objekte und Variablen nach Belieben, wie es ihnen gerade in den Sinn kommt. Bei kleinen Projekten und Programmen mag das vielleicht nicht weiter tragisch sein, wenn man jedoch eine größere Applikation bereitstellen oder im Team an einem Projekt arbeiten möchte, wird diese Willkür schnell zum Albtraum. Da kann es vorkommen, dass man selbst nicht mehr weiß, wofür die Variable jetzt eigentlich gedacht war oder welcher Inhalt in die seltsam benannte Textbox kommt. Schlimmer kann es dann noch werden, wenn man selbst nicht mehr weiß, ob es sich bei dem benutzten Namen um eine Variable oder um ein Objekt handelt. So kann es schnell passieren, dass man einer Variablen einen falschen Wert zuweist oder eine Textbox mit falschen Daten versorgt. Lesen Sie die nächsten Zeilen bitte mit besonderer Sorgfalt. Wenn Sie sich die richtige Namensgebung gleich zu Beginn angewöhnen, werden Sie später keine Sorgen mit den oben geschilderten Problemen bekommen.

Vielleicht haben Sie schon etwas von der ungarischen Konvention gehört. Diese wurde von Charles Simonyi für Programmiersprachen vor der .NET-Zeit eingeführt. Dabei wurde der Gültigkeitsbereich der Variablen sowie der Variablentyp als Präfix geschrieben.

`g_iNummer` war zum Beispiel eine `Integer`-Variable, die global deklariert war.

Mit .NET hat Microsoft sich aber von diesem Standard entfernt und einige Neuerungen im Bereich der Framework-Programmierung hinzugefügt.

Wie Sie vielleicht wissen, ist ein Namespace eine logische Dateneinheit, die für die Organisation von Klassen verwendet wird. Die allgemein gültige Konvention für die Namensgebung von Namespaces ist, den Namen des Unternehmens, von dem der Namespace bereitgestellt wird, an die erste Stelle zu setzen. An zweiter Stelle folgt dann der technologische Name. Die Bereiche werden jeweils mit einem Punkt voneinander getrennt. Nehmen wir zum Beispiel an, ein Unternehmen mit dem Namen *DataService* hat eine Datenbibliothek bereitgestellt. Der richtige Namespace-Name könnte dann Namespace `DataService.Data` lauten.

Sie sollten für Ihre Klassen möglichst »sprechende« Bezeichner vergeben. Außerdem ist zu empfehlen, jeden Anfangsbuchstaben eines neuen Worts groß zu schreiben (Pascal-Notation). Unterstriche sollten vermieden werden. Ein Name für eine Klasse könnte zum Beispiel `StartProcedure` lauten. Ausgehend von unserem oberen Beispiel hieße die Klasse dann `DataService.Data.StartProcedure`. Definiert würde sie im Namespace dann beispielsweise mit `Public Class StartProcedure`.

Für Interfaces sollte ebenfalls ein Bezeichner verwendet werden, der beschreibt, wofür das Interface benötigt wird. Um deutlich zu machen, dass es sich um ein Interface handelt, sollten Interface-Namen immer mit einem großen I beginnen.

```
Interface ICall
```

Als Methode bezeichnet man eine Aktion oder ein Verhalten, das von einem Objekt ausgeführt wird. Hier sollte als Bezeichnung ein Verb verwendet werden. Methoden sollten auch in der Pascal-Notation geschrieben werden.

```
Public Function ShowAll() As Boolean
```

Parameter sollten ebenfalls selbst erklärend sein. Der erste Buchstabe sollte kleingeschrieben werden und jedes weitere Wort im Bezeichner sollte mit einem Großbuchstaben beginnen (camelCasing-Notation).

```
Public Function ShowAll( _
    ByVal firstName As String, _
    ByVal lastName As String)
```

Ein Event wird von einem Objekt als eine Benachrichtigung gesendet. Diese Benachrichtigung kann daraufhin weiterverarbeitet werden (in Kapitel 5 erfahren Sie dazu mehr). Ein EventHandler sollte immer das Suffix `EventHandler` tragen, damit man ihn als solchen identifizieren kann, und in der Pascal-Notation geschrieben sein.

```
Public Delegate Sub ObjectEventHandler
```

Ein Suffix sollte ebenso jedem Attribut mit angehängt werden.

```
Public Class ShowAllAttribute
```

Private Felder, sowie auch lokale Variablen, sollten in der camelCasing-Notation geschrieben werden, während für Eigenschaften die Pascal-Notation zu verwenden ist.

```
Private vorname As String
```

3.1.11 Implizite Typisierung lokaler Variablen

Unter impliziter Typisierung von lokalen Variablen versteht man, dass eine Variable nicht mehr mit einem konkreten Datentyp definiert werden muss, wenn sie bei der Definition einen Wert zugewiesen bekommt.

```
Dim value = 17
```

Der Compiler erkennt aufgrund der Wertzuweisung, dass es sich hierbei um eine Variable vom Typ `Integer` handelt, und führt die Typisierung implizit durch.

Achtung

Bitte denken Sie jetzt keinesfalls, dass nun der Datentyp `Variant` wieder eingeführt worden ist. Implizite Typisierung vergibt den Datentyp automatisch, deswegen muss auch die Zuweisung eines Werts zwingend bei der Definition der Variablen geschehen. Eine Änderung des Datentyps im weiteren Verlauf der Methode, wie es bei `Variant` ja möglich war, ist nicht erlaubt.

Auch IntelliSense unterstützt Sie in Bezug auf implizit typisierte Variablen wie gewohnt, wie Sie in Abbildung 3.3 sehen.

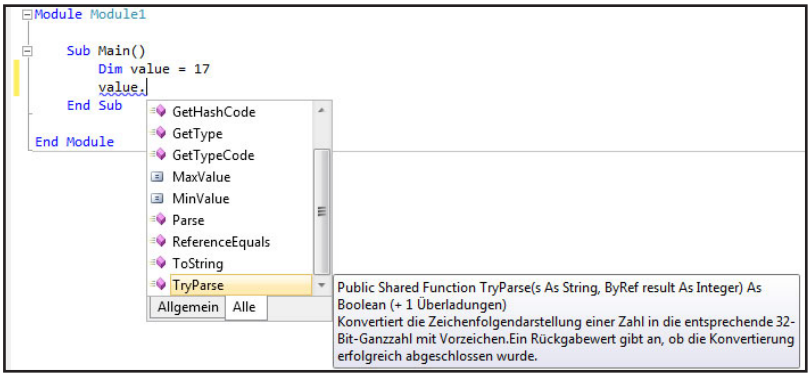

Abbildung 3.3
IntelliSense bei impliziter Typisierung

Implizite Typisierung ist jedoch nur für lokale Variablen erlaubt und eine Zuweisung mit `Nothing` ist ebenfalls nicht gestattet.

Die Variable muss bei der Definition auch initialisiert werden, wobei eine Initialisierung auch durch einen Funktionsaufruf erfolgen kann.

Wenn Sie sich die Mühe machen und einen Blick in den IL-Code werfen – das Tool *ildasm.exe* eignet sich hervorragend dazu – werden Sie sehen, dass die Variable tatsächlich auch als `Integer` definiert ist.

Ich möchte jedoch gleich vorausschicken, dass Sie implizite Typisierung nicht bei so offensichtlichen Definitionen anwenden sollten wie in diesem kleinen Beispiel. Das würde mit Sicherheit nur die Lesbarkeit Ihres Programmcodes verschlechtern und nicht wirklich einen Vorteil bringen.

Sie werden jedoch im Verlauf dieses Buchs noch sehen, dass es aufgrund von LINQ Situationen gibt, bei denen Sie den Rückgabewert einer Funktion gar nicht kennen und unter Umständen auch gar nicht definiert haben. Und genau für solche Fälle wurde die implizite Typisierung eingeführt.

3.2 Kontrollstrukturen

Wie Ihnen bekannt sein wird, werden Befehle in Visual Basic grundsätzlich von oben nach unten, das heißt, Zeile für Zeile abgearbeitet. Der Compiler führt jede Zeile aus und startet erst dann mit der nächsten, wenn der vorige

Vorgang abgefertigt ist. Diese Vorgehensweise können Sie mit sogenannten Verzweigungen ändern. Listing 3.8 demonstriert, wie Verzweigungen im eigentlichen Sinne funktionieren.

3.2.1 If-Else-End If

Listing 3.8
Beispiel für Verzweigungen

```
Public Sub Verzweigung()
    Computer = My.Application.Culture.ToString
    Pruefen(Computer)
    MessageBox.Show("Prüfung abgeschlossen")
End Sub
Sub Pruefen(Computer As String)
    If Computer = "de-DE" Then
        MessageBox.Show(
            "Aktuelle Sprache: Deutsch")
    Else
        MessageBox.Show("Sprache nicht bekannt")
    End If
End Sub
```

In Listing 3.8 wird die aktuelle Spracheinstellung des Computers ermittelt und in einen String geschrieben. Danach wird die `Pruefen()`-Methode aufgerufen (in der Programmierung häufig auch als **Invoking** bezeichnet). Dort wird kontrolliert, ob die aktuelle Spracheinstellung Deutsch ist. Falls das der Fall ist, wird eine Meldung ausgegeben. Nachdem die `Pruefen()`-Methode beendet wurde, kehrt die Programmausführung wieder in die aufrufende Methode zurück und führt dort die nächste Zeile aus, die in diesem Beispiel das Programm beendet.

Info

In Listing 3.8 wurde der `My`-Namespace verwendet. Mehr zu diesem Namespace erfahren Sie in Kapitel 5.8, »Das My-Object«.

Der im obigen Beispiel dargestellte Else-Zweig ist optional. Genauso können Sie mit `ElseIf`- weitere Else-Zweige mit unterschiedlichen Bedingungen hinzufügen.

3.2.2 Der ternäre Operator IIf

Visual Basic stellt mit dem `IIf`-Operator einen ternären Operator zur Verfügung. Der `IIf`-Operator besitzt drei Argumente. Der erste ist ein zu überprüfender Ausdruck und die anderen beiden Argumente sind mögliche Werte, die den Rückgabewert dieses Operators festlegen. Ist der erste Ausdruck wahr, dann wird das zweite Argument zurückgegeben, ansonsten das dritte Argument. Im nachfolgenden Beispiel wird überprüft, ob eine Variable `i` einen Wert größer als 5 besitzt. Ist das der Fall, wird das zweite Argument zurückgegeben, ansonsten das dritte Argument.

```
Dim s As String = IIf(i>5, "groß", "klein").ToString()
```

NEW

Beim `IIf`-Operator wird seit Visual Basic 9 nur das Argument ausgewertet, das tatsächlich auch zurückgegeben wird. In den Vorgängerversionen wurden immer beide möglichen Rückgabewerte ausgewertet.

Die Anweisung

```
Dim i as Integer = Convert.ToInt32(IIf (x Is Nothing, 0,
x.Test))
```

hätte früher, falls x tatsächlich `Nothing` wäre, eine `NullReferenceException` als Laufzeitfehler geworfen, weil die Eigenschaft `Test` (das zugehörige Objekt ist ja `Nothing`) nicht ausgewertet werden konnte. Jetzt wird einfach 0 zurückgegeben.

3.2.3 Select Case

Wenn Sie einen festen Ausdruck mit verschiedenen Alternativen vergleichen wollen, kann es sehr umständlich werden, diesen mit einer Aneinanderreihung von `If`-Ausdrücken (oder auch `ElseIf`-Ausdrücken) zu bewältigen. Praktischer und einfacher geht es mit dem `Select Case`-Ausdruck. Dieser überprüft eine Variable, deren Wert mit den Werten der einzelnen Case-Zweige verglichen wird. Jeder Case-Zweig besitzt einen Anweisungsblock. Bei der Programmausführung kommt derjenige zum Zug, dessen Wert mit dem der Variablen übereinstimmt. In Listing 3.9 sehen Sie, wie eine `Select Case`-Anweisung aufgebaut ist:

Listing 3.9 Select Case-Anweisung

```
Select Case ZuPrüfendeVariable
    Case Wert1
        'Befehl
    Case Wert2
        'Befehl
    Case Wert2
        'Befehl
    Case Else
        'Befehl
End Select
```

Der `Case Else`-Block wird ausgeführt, wenn in der vorhergehenden Liste kein Wert auf den zu prüfenden Wert passt. Der `Case Else`-Block ist optional.

Dabei ist es auch möglich, statt eines konkreten Werts einen Wertebereich anzugeben, wie in Listing 3.10 dargestellt.

Listing 3.10 Select Case-Anweisung mit Werteblöcken

```
Select Case ZuPrüfendeVariable
    Case 1 To 5
        'Befehl
    Case 6 To 20
        'Befehl
    Case 21 To 50
        'Befehl
    Case Else
        'Befehl
End Select
```

3.3 Schleifen

Schleifen dienen in der Regel dazu, Befehle so lange zu wiederholen, bis eine bestimmte Bedingung eingetreten ist. Dieser Vorgang wird zu Beginn oder am Ende der Schleife festgelegt und bei jedem Schleifendurchlauf am Anfang oder Ende geprüft. Man spricht deswegen auch von kopf- und fußgesteuerten Schleifen.

3.3.1 Do Loop-Schleife

Die `Do Loop`-Schleife kann in verschiedenen Varianten auftreten. Sie wird dazu verwendet, so lange einen Prozess zu wiederholen, bis eine bestimmte Voraussetzung erfüllt oder nicht mehr erfüllt ist.

Kopfgesteuerte Schleife:

```
Dim i As Integer = 0
Do Until i > 100
    i += Convert.ToInt32(Console.ReadLine())
Loop
```

Fußgesteuerte Schleife:

```
Dim i As Integer = 0
Do
    i += Convert.ToInt32(Console.ReadLine())
Loop Until i > 100
```

Die fußgesteuerte Schleife wird mindestens ein Mal durchlaufen, weil die erste Prüfung erst am Ende der Schleife erfolgt. Die kopfgesteuerte Schleife kann unter Umständen auch kein einziges Mal durchlaufen werden.

3.3.2 Die While-Schleife

Die `While`-Schleife ist sehr ähnlich zur `Do Loop`-Schleife mit dem Unterschied, dass bei der `While`-Schleife die Schleife so lange ausgeführt wird, wie die Bedingung `true` ist.

```
Dim i As Integer
While i < 100
    Console.WriteLine(i)
    i += Convert.ToInt32(Console.ReadLine())
End While
Console.ReadLine()
```

Alternativ können Sie das Schlüsselwort `While` auch in einer `Do Loop`-Schleife verwenden. Statt `Until` wird dann einfach das Schlüsselwort `While` eingesetzt.

```
Dim i As Integer = 0
Do While i < 100
    i += Convert.ToInt32(Console.ReadLine())
Loop
```

3.3.3 Die For Next-Schleife

Wenn Sie bereits wissen, wie oft die Schleife durchlaufen werden soll, empfiehlt sich die `For Next`-Schleife. Bei der Schleife in Listing 3.11 wird die Schleife von 10 bis 100 durchlaufen und der Wert jeweils in der Konsole ausgegeben.

Listing 3.11 For Next-Schleife im Einsatz

```
For i As Integer = 10 To 100
    Console.WriteLine(i)
Next
Console.ReadLine()
```

Optional können Sie auch mit dem Schlüsselwort `Step` die Schrittweite beim Schleifendurchlauf verändern. Standardmäßig ist die Schrittweite +1.

```
For i As Integer = 100 To 10 Step -1
    Console.WriteLine(i)
Next
```

3.3.4 Die For Each-Schleife

Die `For Each`-Schleife wird am häufigsten für das Durchlaufen von Arrays oder Collections benutzt. Hierbei wird nicht wie bei der `For Next`-Schleife ein Zahlenwert hochgezählt, sondern auf ein Element innerhalb einer Liste zurückgegriffen. Die Schleife wird so lange wiederholt, wie Elemente in der Auflistung vorhanden sind.

```
For Each c As Control In Me.Controls
    MessageBox.Show(c.Name)
Next
```

Die `For Each`-Schleife kann über alle aufzählbaren Objekte iterieren.

3.4 Arrays

In Visual Basic 10 gibt es zwei Arten von Arrays. Zum einen Arrays, die einfach mit Klammern deklariert werden. In diesem Fall sind Arrays Variablen, die Werte des gleichen Datentyps speichern können. Zur besseren Veranschaulichung können Sie sich eine Liste vorstellen, die zunächst eine Nummer und dann den dazugehörigen Wert, wie in Tabelle 3.5, enthält.

Tabelle 3.5 Werte in einem Array

Index	Wert
0	Kotz
1	Haban
2	Mühlbauer
3	Klaubert
4	Saucke
5	Graupp
6	Zauner
7	Schuldt

Arrays beginnen – im Gegensatz zur früheren Visual Basic 6-Version – immer mit dem Index 0. Die Index-Obergrenze kann bei der Deklaration in Klammern hinter dem Array-Namen angegeben werden, zum Beispiel:

```
Dim ArrayName(99) As String
```

Hier enthält das Array 100 Werte, da der Index des ersten Elements 0 ist.

Achtung

Zum Vergleich: In C# geben Sie in den Klammern die Anzahl der Elemente an.

Es ist nicht unbedingt erforderlich, die Anzahl der Elemente bei der Deklaration anzugeben. Diese kann auch später durch `ReDim` festgelegt werden, falls die Anzahl der Elemente noch nicht feststeht. Beachten Sie jedoch, dass `ReDim` sehr viel Rechenzeit benötigt.

Im folgenden Listing 3.12 wird gezeigt, wie man ein Array mit Werten füllt und dieses dann in einer Konsole ausgeben kann.

Listing 3.12
Ausgabe von Arrays in einer Konsolenanwendung

```
Sub Main()
    Dim aZahlenbeispiele(9) As Integer
    Dim i As Integer
    aZahlenbeispiele(0) = 1
    aZahlenbeispiele(1) = 2
    aZahlenbeispiele(2) = 3
    aZahlenbeispiele(3) = 4
    aZahlenbeispiele(4) = 5
    aZahlenbeispiele(5) = 6
    aZahlenbeispiele(6) = 7
    aZahlenbeispiele(7) = 8
    aZahlenbeispiele(8) = 9
    aZahlenbeispiele(9) = 10
    For i = 0 To aZahlenbeispiele.Length - 1
        Console.WriteLine(aZahlenbeispiele(i))
    Next
    Console.ReadLine()
End Sub
```

Die Ausgabe würde dann in der integrierten Konsole von Visual Studio 2010 aussehen wie in Abbildung 3.4.

Info

Die Ausgabe der Array-Werte erfolgt hier in einer Konsolenanwendung. Konsolenanwendungen lassen sich mit jeder Visual Studio 2010-Version erstellen und dienen zum Testen der Anwendung in einem Konsolenfenster ähnlich wie unter MS DOS. Eine neue Konsolenanwendung legen Sie wie folgt an: Klicken Sie im Menü auf Datei/Neu/Projekt, und wählen Sie dann unter Visual Basic-Projekte die Vorlage Konsolenanwendung aus. Sie können jetzt innerhalb der *Sub Main*-Prozedur Ihren Code entwickeln. Dieser wird beim Start des Projekts ausgeführt.

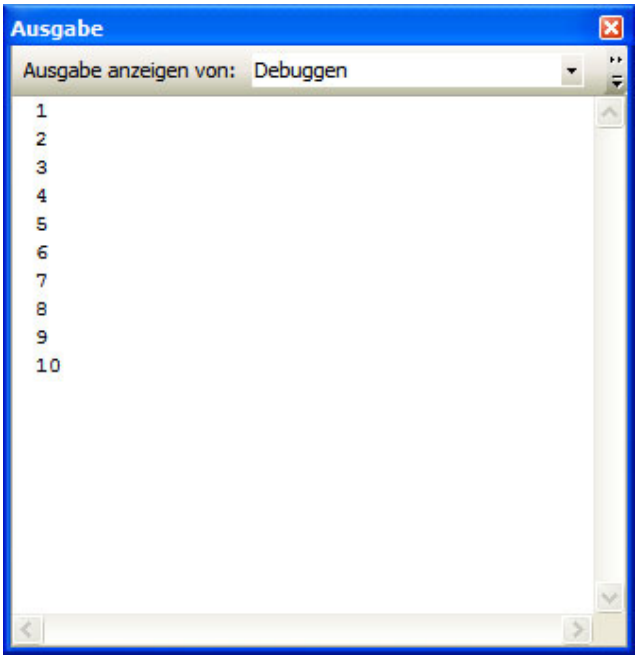

Abbildung 3.4 Ausgabe des Array-Beispiels in der Visual Studio-Konsole

Zum anderen kann ein Array aber auch als Objekt behandelt werden. Hierbei wird das Array bei der Deklaration mit der `CreateInstance`-Methode bekannt gemacht. Die Werte des Arrays müssen mit `SetValue` zugewiesen und mit der `GetValue`-Methode wieder ausgelesen werden. Zur Verdeutlichung wird Listing 3.12 umgeschrieben. Das Ergebnis sehen Sie in Listing 3.13.

Listing 3.13 Ausgabe des Array-Objekts

```
Public Shared Sub Main()
    Dim aZahlenbeispiele As Array = _
        Array.CreateInstance( _
        GetType(Integer), 11)

    aZahlenbeispiele.SetValue(1, 0)
    aZahlenbeispiele.SetValue(2, 1)
    aZahlenbeispiele.SetValue(3, 2)
    aZahlenbeispiele.SetValue(4, 3)
    aZahlenbeispiele.SetValue(5, 4)
    aZahlenbeispiele.SetValue(6, 5)
    aZahlenbeispiele.SetValue(7, 6)
    aZahlenbeispiele.SetValue(8, 7)
    aZahlenbeispiele.SetValue(9, 8)
    aZahlenbeispiele.SetValue(10, 9)
    For i As Integer = 0 To _
        aZahlenbeispiele.Length - 1

        Console.WriteLine _
          (aZahlenbeispiele.GetValue(i) _
           .ToString)
    Next
    Console.ReadLine()
End Sub
```

In der Praxis finden Sie zumeist die erste hier vorgestellte Version von Arrays im Einsatz.

3.4.1 Mehrdimensionale Arrays

Sowohl »gewöhnliche« Arrays als auch Array-Objekte können mehrdimensionale Werte beinhalten. Das folgende Beispiel (Listing 3.14) zeigt ein vierdimensionales Array, das in der ersten Position fünf, in der zweiten zehn, in der dritten 100 und in der vierten wiederum fünf Elemente aufweist. Nach der Vereinbarung (als gewöhnliches Array sowie als Array-Objekt) erfolgt in der nächsten Zeile jeweils die Wertzuweisung an ein einzelnes Element.

Listing 3.14
Mehrdimensionale Arrays

```
Public Shared Sub Main()
    Dim Werte(4, 9, 99, 4) As Integer
    Werte(1, 2, 3, 4) = 500
    'Oder mit dem Array-Objekt
    Dim NeueWerte As Array = _
    Array.CreateInstance(GetType(Integer), _
        4, 9, 99, 4)
    NeueWerte.SetValue(500, 1, 2, 3, 4)
End Sub
```

Das oben definierte Array kann im Übrigen 25.000 Elemente aufnehmen (5*10*100*5).

3.4.2 Arrays hinzufügen

Oftmals wissen Sie während der Programmierung nicht, wie viele Werte ein Array später enthalten wird (stellen Sie sich einmal vor, Sie lesen den gesamten Inhalt eines Verzeichnisses in ein Array, können aber vorher nicht bestimmen, wie viele Dateien in diesem Verzeichnis abgelegt sind). Hier schafft die `ReDim`-Anweisung Abhilfe.

Um die bereits gespeicherten Werte nicht zu verlieren, sollten Sie dabei das Schlüsselwort `Preserve` wie in Listing 3.15 benutzen.

Listing 3.15
Arrays mit ReDim erweitern

```
Public Sub Addmore()
    Dim aWerte(9) As Integer
    For i As Integer = 0 To 9
        aWerte(i) = i + 1
    Next i
    'ReDim, Werte bleiben erhalten
    ReDim Preserve aWerte(19)
    For i As Integer = 10 To 19
        aWerte(i) = i + 1
    Next i

    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.ReadLine()
End Sub
```

Tipp

Ohne dem weiteren Verlauf dieses Buchs vorgreifen zu wollen, möchte ich darauf hinweisen, dass bei so einer Aufgabenstellung Collections die bessere Wahl sind.

3.4.3 Arrays sortieren

Mit der `Array.Sort()`-Methode können Sie die Werte eines Arrays sortieren. Die Methode vergleicht dabei jeden einzelnen *Array*-Wert mit den anderen. Das Beispiel in Listing 3.16 erstellt mit der `Random`-Klasse zehn zufällige Werte zwischen 1 und 100 und legt diese Werte in dem Array `aWerte` ab. Danach werden die unsortierten Werte ausgegeben, bevor sie sortiert und dann erneut ausgegeben werden.

Listing 3.16 Arrays sortieren

```
Public Sub Sortieren()
    Dim aWerte(9) As Integer
    Dim myrand As New Random()
    For i As Integer = 0 To 9
        aWerte(i) = myrand.Next(1, 100)
    Next i
    'Unsortierte Werte ausgeben
    Console.WriteLine("Unsortierte Werte:")
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.WriteLine(vbNewLine & _
        "Nun folgen die sortierten Werte:")
    'Array sortieren
    Array.Sort(aWerte)
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
End Sub
```

Die Ausgabe könnte so aussehen:

```
Unsortierte Werte:
11
64
8
74
47
10
78
52
66
51
Nun folgen die sortierten Werte:
8
10
11
47
51
52
64
66
74
78
```

3.4.4 Arrays invertieren

Mit der `Array.Reverse()`-Methode können Sie geordnete Arrays in umgekehrter Reihenfolge ordnen. Das Beispiel in Listing 3.17 zeigt, wie Sie einem Array zunächst Werte von 1 bis 10 in aufsteigender Reihenfolge hinzufügen und diese dann umkehren.

Listing 3.17
Arrays umkehren

```
Public Sub Reverse()
    Dim aWerte(9) As Integer
    For i As Integer = 0 To 9
        aWerte(i) = i + 1
    Next i
    Array.Reverse(aWerte)
    For i As Integer = 0 To aWerte.Length - 1
        Console.WriteLine(aWerte(i))
    Next
    Console.ReadLine()
End Sub
```

3.4.5 Arrays durchsuchen

Nachdem das Array nun sortiert wurde, kann dieses auch durchsucht werden. Der Suchalgorithmus arbeitet auch noch bei großen Arrays sehr schnell und bietet so die optimale Möglichkeit, nach bestimmten Daten zu suchen. Nachdem die Werte in dem Array abgelegt wurden, müssen diese mit der `Array.Sort()`-Methode sortiert werden, danach kann über `Array.BinarySearch()`der Index der zu suchenden Person ermittelt werden. Am Ende der Beispielanwendung in Listing 3.18 werden der neue Index nach der Sortierung und der Name der Person ausgegeben.

Listing 3.18
Array nach bestimmtem Wert durchsuchen

```
Public Sub Suchen()
    Dim aPersonen(7) As String
    Dim zuSuchendePerson As String
    Dim Result As String
    Dim personIndex As Integer
    aPersonen(0) = "Kotz"
    aPersonen(1) = "Haban"
    aPersonen(2) = "Klaubert"
    aPersonen(3) = "Mühlbauer"
    aPersonen(4) = "Graup"
    aPersonen(5) = "Bremer"
    aPersonen(6) = "Zauner"
    aPersonen(7) = "Albert"
    Array.Sort(aPersonen)
    zuSuchEndePerson = "Graup"
    personIndex = Array.BinarySearch(aPersonen, _
        zuSuchendePerson)
    Result = aPersonen(personIndex)
    MessageBox.Show _
        ("Die zu suchende Person wurde im " & _
         "Index an der neuen Stelle " & _
         personIndex & _
         " gefunden. Die Person ist: " & Result, _
         "Suche in Arrays", _
         MessageBoxButtons.OK, _
         MessageBoxIcon.Information)
End Sub
```

Die Ausgabe sehen Sie in Abbildung 3.5.

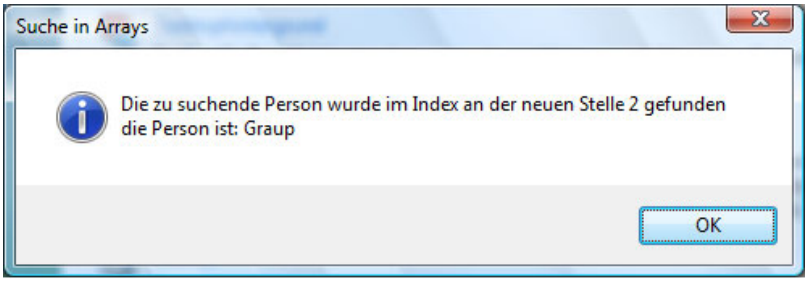

Abbildung 3.5
Ergebnis der Array-Durchsuchung

3.5 Operatoren

Operatoren verknüpfen im eigentlichen Sinn zwei Ausdrücke miteinander. Diese Ausdrücke können aus Zeichenketten (Strings), Zahlen oder Formeln bestehen. Bei der Programmierung unterscheiden wir grundsätzlich unterschiedliche Arten von Operatoren.

- Arithmetische Operatoren dienen dazu, Berechnungen durchzuführen.
- Verknüpfungsoperatoren verknüpfen unterschiedliche Ausdrücke miteinander.
- Logische Operatoren werden zur bitweisen Verknüpfung von Operanden benötigt.
- Vergleichsoperatoren vergleichen unterschiedliche Ausdrücke miteinander.

Die arithmetischen Operatoren in Tabelle 3.6 können bei der Programmierung mit Visual Basic 10 verwendet werden:

Tabelle 3.6
Operatoren im Überblick

Arithmetischer Operator	Funktion
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
\	Integerdivision
^	Potenzierung
Mod	Bestimmung des Restwerts

Die Darstellung der Grundrechenarten ist in Visual Basic denkbar einfach. Hier einige Beispiele für die Verwendung von Grundrechenarten:

Beispiel für Addition:

```
dblAusgabe = 10 + 3
```

Beispiel für Subtraktion:

```
dblAusgabe = 10 - 3
```

Beispiel für Multiplikation:

```
dblAusgabe = 10 * 3
```

Beispiel für Division:

```
dblAusgabe = 10 / 3
```

Die Integerdivision liefert im Gegensatz zur Division keine gebrochenen Zahlen, sondern einen ganzzahligen Wert.

Achtung

Seien Sie bitte nicht überrascht, dass es bei einer Gleitkommazahlendivision (mit /) bei der Division durch 0 keinen Fehler gibt. Es wird mit einer Annäherung von 0 dividiert.

Anstelle des Operators / wird der Backslash \ eingesetzt.

```
dblAusgabe = 10 \ 3
```

Im Gegensatz zur Integerdivision liefert der Mod-Operator den Rest einer Division.

```
dblAusgabe = 10 Mod 3
```

Bei der Potenzierung wird nicht 10^3 geschrieben, sondern das Zeichen ^ verwendet.

```
dblAusgabe = 10 ^ 3
```

Visual Basic kennt seit der Sprachversion 8, die mit Visual Studio 2005 veröffentlicht wurde, auch zusammengesetzte Zuweisungsoperatoren. Anstatt beim Hochzählen einer Variablen beispielsweise `i = i + 1` zu schreiben, können Sie den Shortcut `i += 1` verwenden. Dasselbe gilt für den Subtraktions-, Multiplikations- und Divisionsoperator.

Verknüpfungsoperatoren werden immer dann gebraucht, wenn Sie in einer Ausgabe einen String mit einem anderen String oder einem anderen Datentyp verwenden wollen. Für die Verbindung wird das Kaufmannsund (&) als Verknüpfungsoperator benötigt.

```
MessageBox.Show("Der Wert der Zahl beträgt " & _
    intZahl1.ToString)
```

Logische Operatoren dienen dazu, mehrere Teilbedingungen miteinander zu verknüpfen und damit zum Beispiel Wahrheitsüberprüfungen festzustellen.

Visual Basic kennt die in Tabelle 3.7 aufgelisteten logischen Operatoren.

Tabelle 3.7
Logische Operatoren

Operator	Funktion
And	Und-Verknüpfung
Or	Oder-Verknüpfung
Not	Negation

Tabelle 3.7 (Forts.) Logische Operatoren

Operator	Funktion
Xor	Exklusive Oder-Verknüpfung
AndAlso	Und-Verknüpfung, die jedoch die Überprüfung aller Ausdrücke abbricht, falls das Ergebnis eindeutig ist
OrElse	Oder-Verknüpfung, die jedoch die Überprüfung aller Ausdrücke abbricht, falls das Ergebnis eindeutig ist

Beispiel für den `And`-Operator:

```
If intZahl1 > 1 And intZahl2 < 100 Then
MessageBox.Show("Der gesuchte Wert liegt zwischen 1 und 100")
```

Beispiel für den `Or`-Operator:

```
If intZahl1 < 1 Or intZahl2 > 100 Then
MessageBox.Show("Der Wert liegt unter 1 oder über 100")
```

Der `Xor`-Operator stellt eine Besonderheit dar. Bei `Xor` wird ein logischer Ausschluss zweier boolescher oder ein bitweiser Ausschluss zweier numerischer Ausdrücke durchsucht.

```
If (intZahl1 = 2 Xor 8) > 0 Then
MessageBox.Show("Der Ausdruck ist wahr").
```

Der `Not`-Operator kehrt den Wert des Ausdrucks um. Wenn also beispielsweise das Ergebnis einer If-Abfrage `True` ist, ändert der `Not`-Operator den Wert zu `False`, andersherum ändert er den `False`-Wert zu `True`.

Vergleichsoperatoren werden häufig bei `If`-Anweisungen benutzt, um Variablen und/oder Werte miteinander zu vergleichen.

Tabelle 3.8 zeigt die in Visual Basic 9 verwendbaren Vergleichsoperatoren.

Tabelle 3.8 Vergleichsoperatoren

Operator	Funktion
=	Gleich
<	Kleiner
>	Größer
<=	Kleiner gleich
>=	Größer gleich
<>	Ungleich

In Listing 3.19 sehen Sie, wie man Zahlenwerte miteinander vergleicht. Hierbei spielt es keine Rolle, ob die Zahl als Wert in einer Variablen gespeichert ist oder direkt in der Anweisung steht.

Listing 3.19 Vergleich von Zahlenwerten

```
Dim intZahl1 As Integer = 5
Dim intZahl2 As Integer = 10
If intZahl1 > intZahl2 Then
    MessageBox.Show(
      "Zahl 1 ist größer als Zahl 2")
ElseIf intZahl1 < intZahl2 Then
```

```
    MessageBox.Show(
      "Zahl 1 ist kleiner als Zahl 2")
ElseIf intZahl1 = intZahl2 Then
    MessageBox.Show(
      "Zahl 1 ist gleich als Zahl 2")
End If
```

Listing 3.19 (Forts.) Vergleich von Zahlenwerten

3.6 Befehle

Komplette Programmanweisungen werden in Visual Basic 10 Befehle genannt. Jeder Befehl endet mit einer neuen Zeile oder einem Doppelpunkt.

```
Dim Text As String = "hallo"
Dim T2 As String = "w" : Dim i As Integer = "1"
```

Hierbei wird der Doppelpunkt oft als unprofessionelle Art der Programmierung gesehen, da es die Lesbarkeit der Anwendung um einiges erschwert.

Falls eine Anweisung sehr lang ist, können Sie im Sinne einer besseren Lesbarkeit den Unterstrich (_) benutzen, um einen Befehl auf mehrere Zeilen zu verteilen:

```
MessageBox.Show("Hallo, was ich schon immer mal …", _
    "Titelüberschrift")
```

Vor dem Unterstrich muss zwingend ein Leerzeichen stehen.

Mit Visual Basic 10 kann nun auf die Verwendung des Unterstrichs verzichtet werden.

3.7 Sonstige Sprachelemente

An dieser Stelle will ich Ihnen noch einen Überblick über weitere wichtige Sprachelemente und Schlüsselwörter geben.

3.7.1 Die Operatoren Is und IsNot

Objektvariablen vergleichen Sie mit den Operatoren `Is` und `IsNot`.

```
If obj1 Is obj2 Then
```

Die obige If-Bedingung ist wahr, wenn beide Variablen auf dasselbe Objekt verweisen.

Wenn es darum geht, zu prüfen, ob eine Objektvariable überhaupt einen Verweis auf ein Objekt enthält, erweist sich der Einsatz des Operators `IsNot` oft als eleganter. Die Codezeile

```
If obj IsNot Nothing Then
```

liest sich vermutlich intuitiver als

```
If Not obj Is Nothing Then.
```

3.7.2 Weitere Schlüsselwörter

Folgende drei Schlüsselwörter, die mit Visual Basic 8 eingeführt worden sind, wollen wir uns hier noch kurz ansehen:

Global

Bevor ich Ihnen das Schlüsselwort `Global` näher erläutere, erlauben Sie mir vorab eine kleine Bemerkung. Ich hoffe nur, dass Sie dieses Schlüsselwort nie benötigen. Ich bin zumindest die letzten fünf Jahre darum herum gekommen.

Denn tatsächlich benötigen Sie es nur, wenn Sie bereits gravierende Namenskonflikte haben. Stellen Sie sich vor, Sie haben eigene Namespaces definiert und wollen diese auch so sprechend wie möglich benennen. Ein sehr löbliches Vorgehen, sei noch kurz angemerkt.

Nun heißt einer von diesen Namespaces zufällig `FirmenName.GlobalClasses.System.Data`.

Und schon haben Sie ein Problem, wenn Sie eine Klasse aus dem Original-Namespace `System.Data` verwenden wollen, denn dieser wird jetzt durch Ihren eigenen Namespace überschattet.

Wenn Sie in Ihrem eigenen Namespace nun folgende Variablendefinition vornehmen:

```
Dim ds As New System.Data.DataSet
```

wird diese nicht funktionieren, da der Compiler eine Klasse `DataSet` in Ihrem eigenen Namespace sucht.

Beheben können Sie das mit dem Schlüsselwort `Global`, indem Sie die Definition folgendermaßen durchführen:

```
Dim ds as New Global.System.Data.DataSet
```

Using

Da wir im .NET Framework ein nichtdeterministisches Verhalten des Garbage Collector haben, können wir nicht vorhersehen, zu welchem Zeitpunkt unsere Objekte wieder im Speicher freigegeben werden. Der Zeitpunkt liegt zwar in der nahen Zukunft, aber unsere Objekte halten so lange noch Zugriff auf Ressourcen, die sie beansprucht haben. Um diese Ressourcen bereits vorzeitig freizugeben, können Sie für Ihre Objekte das `IDisposable`-Interface implementieren und eine Methode `Dispose` bereitstellen, aber Sie können nicht beeinflussen, dass der Entwickler, der Ihre Objekte nutzt, die Methode `Dispose` auch aufruft.

Sie können nun Objekte innerhalb eines `Using`-Blocks definieren. Dadurch wird die Laufzeitumgebung automatisch am Ende des Blocks die entsprechende `Dispose`-Methode aufrufen. Dadurch ist aber auch klar, dass Sie `Using` nur für Objekte benutzen können, die das `IDisposable`-Interface implementieren.

Das Schlüsselwort `Using` verwenden Sie dabei wie im folgenden Beispiel dargestellt:

```
Using (a As New Artikel)
    'beliebiger Code
End Using
```

Eine Verwendung der Objektvariablen ist nach dem Ende des `Using`-Blocks nicht mehr möglich.

Wobei Sie bei dieser Syntaxvariante mehrere Objektvariablen für `Using` angeben können:

```
Using a As New Artikel, b As New Artikel
```

Continue

Das `Continue`-Schlüsselwort können Sie dazu verwenden, bestimmte Anweisungen innerhalb einer Schleife zu überspringen.

Achtung

Verwechseln Sie diese Anweisung bitte nicht mit dem vorzeitigen Beenden der Schleife. Es wird lediglich ans Schleifenende gesprungen und dann die nächste Iteration ausgeführt.

Wenn Sie zum Beispiel Datensätze aus einer Datei oder einem `DataSet` durchlaufen und in Abhängigkeit von einem Wert soll eine Verarbeitung nicht stattfinden, können Sie hier mit `Continue` den verarbeitenden Block überspringen. Es wird dann ans Schleifenende gesprungen und der nächste Datensatz wird verarbeitet. Listing 3.20 zeigt eine Verwendung von `Continue`.

Listing 3.20
Verwenden von Continue

```
For Each x As Artikel In Produkte
    If x.Preis > 50 Then
        Continue For
    End If
    'Ansonsten weiterer Code
Next
```

Das Schlüsselwort `Continue` können Sie für jeden Schleifentyp verwenden.

3.8 Konsolenanwendungen erstellen

Visual Basic 10 ermöglicht nicht nur die Programmierung von grafischen Windows-Anwendungen, sondern auch die von sogenannten Konsolenanwendungen. Diese laufen beim Aufruf in der Eingabeaufforderung (der Windows Command Shell) und führen Ein- und Ausgaben über die Standardgeräte `StdIn` (Tastatur) und `StdOut` (Eingabeaufforderungsfenster) durch und dienen als Grundlage für viele Visual Basic 10-Beispiele, die Sie im Internet finden können. Im Mittelpunkt steht die `Console`-Klasse, die ein Eingabeaufforderungsfenster repräsentiert. Mit dem .NET Framework 2.0 wurde die `Console`-Klasse um zahlreiche Mitglieder erweitert, mit der sich unter anderem die Hintergrund- und die Schriftfarbe und die Größe des Konsolenfensters abfragen und einstellen lassen. Lesen Sie hier, welche Möglichkeiten es bei Konsolenanwendungen gibt und wie Sie diese für Ihre eigenen

Anwendungen nutzen können. Um eine Konsolenanwendung zu erstellen, müssen Sie zunächst ein neues Projekt anlegen, um dann die Vorlage KONSOLENANWENDUNG auszuwählen. Wie Sie sehen können, wird nicht wie gewohnt ein Windows-Formular erzeugt, sondern nur eine *.vb*-Datei mit dem Code-Inhalt, den Sie in Listing 3.21 sehen.

Listing 3.21 Code beim Anlegen einer Konsolenanwendung

```
Module Module1
    Sub Main()
    End Sub
End Module
```

Zwischen der Anweisung `Sub Main()` und `End Sub` spielt sich die gesamte Konsolenanwendung ab. Um einen Text auszugeben, können Sie die Methode `Console.WriteLine()` verwenden:

```
Console.WriteLine("Willkommen in der Anwendung")
Console.WriteLine()
```

Mit der Methode `Console.ReadLine()` können Sie den Anwender zur Eingabe auffordern.

Listing 3.22 Einfache Konsolenanwendung

```
Console.WriteLine("Geben Sie Ihren Namen ein:")
Dim Name As String = Console.ReadLine
Console.WriteLine()
Console.Write("Hallo " & Name & _
    " bitte geben Sie den Titel ein:")
Console.WriteLine()
Console.Title = Console.ReadLine
Console.WriteLine()
Console.WriteLine()
```

Die Hintergrund- und die Schriftfarbe der Konsole lassen sich ebenfalls bestimmen.

Listing 3.23 Setzen von Farben für die Konsole

```
Console.WriteLine("Geben Sie eine Farbe ein " _
    " (z.b. red, green, black):")
Console.WriteLine()
Try
    Dim bgkFarbe As String = Console.ReadLine
    Console.BackgroundColor = _
        CType(System.Enum.Parse( _
        GetType(ConsoleColor), _
        bgkFarbe, True), ConsoleColor)
Catch ex As Exception
    Console.Write _
        ("Farbe konnte nicht gefunden werden")
End Try
Console.Write( _
    "Geben Sie die Farbe der Schrift ein:")
Try
    Dim fontFarbe As String = Console.ReadLine
    Console.ForegroundColor = _
        CType(System.Enum.Parse( _
            GetType(ConsoleColor), _
            fontFarbe, True), ConsoleColor)
Catch ex As Exception
    Console.Write _
        ("Farbe konnte nicht gefunden werden")
End Try
```

Die Größe des Konsolenfensters können Sie mit `Console.SetWindowSize(x, y)` festlegen. Zudem können Sie noch die Position des Cursors bestimmen und ihn auf Wunsch verbergen.

```
Console.WriteLine("Cursor unsichtbar machen")
Console.CursorVisible = False
Console.WriteLine()
Console.WriteLine("Weiter geht es mit Enter")
Console.ReadLine()
Console.CursorVisible = True
```

Die Konsole ist neuerdings auch musikalisch (okay, zugegeben, nur der interne Systemlautsprecher kann benutzt werden).

Listing 3.24
Musik in der Konsole

```
Dim Frequenz As Integer
Console.Write("Bitte Frequenz eingeben, " & _
    diese sollte zwischen 37 und 32767 liegen")
Frequenz = Convert.ToInt32 (Console.ReadLine)
Console.WriteLine()
Dim Länge As Integer
Console.Write("Geben Sie die Abspiellänge ein: ")
Länge = Convert.ToInt32(Console.ReadLine)
Console.Beep(Frequenz, Länge)
Console.WriteLine()
Console.WriteLine("Enter beendet die Demo")
Console.ReadLine()
```

Zwar sieht das Ganze nach Spielerei aus, die Konsole kann Ihnen aber bei der alltäglichen Programmierarbeit praktisch zur Seite stehen. Sie können beispielsweise Codefragmente bzw. die Ausgabe von Daten einfach in der Konsole testen und müssen dafür nicht extra Ihre gesamte Anwendung kompilieren.

Außerdem werden Konsolenanwendungen auch sehr gerne bei kleinen Programmen für administrative Aufgaben eingesetzt.