

MASTER  
CLASS

→ Jürgen Kotz

# Visual Basic 2008

Einstieg für Anspruchsvolle

inkl. Lerntest und  
Beispiele auf CD



ADDISON-WESLEY

[ in Kooperation mit ]

PEARSON  
Studium

# 3

## Visual Basic 9 allgemein

Ein Schwerpunkt dieses Buchs liegt darin, die wichtigsten sprachlichen Merkmale von Visual Basic 9 aufzugreifen und Ihnen näherzubringen, wie man mit diesen umfangreichen 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 9 vermitteln.

### 3.1 Datentypen

Variablen gehören zu den wichtigsten Bestandteilen jeder 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 9 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"
```

**Tabelle 3.1**  
Datentypen im  
Überblick

Visual Basic 2005 kennt die in Tabelle 3.1 aufgelisteten Datentypen:

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

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 9 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 strText As String = "Hallo Welt"
```

In diesem Beispiel ist `strText` 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 `strText`.

### Hinweis

`Dim` ist die Abkürzung für Dimension. Diese Abkürzung existiert beispielsweise auch in der Mathematik oder Physik und 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 9 in Abhängigkeit vom Datentyp Standardwerte vergeben. Diese können Sie der Tabelle 3.2 entnehmen:

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

**Tabelle 3.2**  
Initialwerte von  
Datentypen

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

### 3.1.2 Werttyp- 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.

### 3 Visual Basic 9 allgemein

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

Bei den Wertetypen wird der Wert der Variablen direkt an dieser 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.

#### Werttypsemantik

Bei der Zuweisung eines Wertetyps wird eine Kopie der entsprechenden Variablen gemacht. 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 Werttypsemantik 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: den Wert 7 für `zahl1` und den Wert 9 für `zahl2`. Sie sehen, dass bei der Zuweisung `zahl2 = zahl1` eine Kopie des Werts angefertigt wurde.

#### Referenztypsemantik

Bei der Zuweisung eines Referenztyps hingegen wird keine Kopie der entsprechenden Variablen gemacht. 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 Variable `p1` vom Typ `Person` definiert und instanziiert (ohne der Objektorientierung vorgreifen zu wollen, sehen Sie die Definition dieses Typs in Listing 3.1). 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 Wertesemantik. Die Ausnahme wurde von den Framework-Entwicklern gemacht, um die Verarbeitung dieses doch wesentlichen Datentyps intuitiver zu implementieren.

### 3.1.3 Nullable Typen

Die eben 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-Equivalent besitzen, das auch den Wert `Nothing` annehmen kann.

Einen *Nullable Integer* definieren Sie dabei wie folgt:

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

#### Hinweis

Mit der Einführung von 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 Zuweisung von Konstanten sehen Sie in Listing 3.2:

**Listing 3.2**  
Festlegung von  
Konstanten

```
Const intSTUNDENLOHN As Integer = 23  
Const intARBEITSSTUNDEN As Integer = 8  
Const intLOHNPROTAG As Integer = _  
    intSTUNDENLOHN * intARBEITSSTUNDEN  
Const strWindowsPfad 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 den Enumerationswert verwenden, natürlich mit IntelliSense-Unterstützung. Das

hat den Vorteil, dass Sie sich die Bedeutung der Indizes nicht merken müssen und auch die Lesbarkeit des Quellcodes erhöht sich.

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

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

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

**Listing 3.3**  
Verwendung  
einer Enum

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

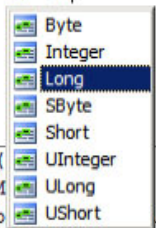
Falls Sie ermitteln wollen, welches Element sich hinter einem Wert in der Aufzählung verbirgt, lässt sich das recht simpel bewerkstelligen, indem Sie an der Enumerationsvariablen die ToString()-Methode aufrufen:

```
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.

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

Public Sub MitarbeiterEnum()
    Dim mitarbeiter As KatM
    mitarbeiter = KatMitarb
    MsgBox(mitarbeiter)
End Sub
End Class
```



**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.



### 3 Visual Basic 9 allgemein

```
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 9 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 9 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 dieses die *Option Strict On*-Eigenschaft nicht erlauben (zu *Option Strict On* erfahren Sie im späteren Teil dieses Kapitels mehr). Daher wäre es jetzt ratsam, diese Variable in einen *Integer*-Wert zu konvertieren. Visual Basic 9 leistet dieses, wie Sie in Listing 3.5 sehen:

```
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)
```

**Listing 3.5**

Konvertierung  
in verschiedene  
Datentypen

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:

Methoden	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

**Tabelle 3.4**

Konvertierungs-  
möglichkeiten

**Tabelle 3.4 (Forts.)**  
Konvertierungsmöglichkeiten

Methode	Konvertierung in folgendes Format
Convert.ToInt64()	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 macht der Compiler automatisch, man spricht dabei von einer impliziten Typumwandlung.

### 3.1.7 Die CType-Methode

Eine weitere Möglichkeit zur Umwandlung von Datentypen bietet Ihnen in Visual Basic 9 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.

**Dim Text As Object**

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

**Listing 3.6**

Verwendung  
von GetType()

### 3.1.9 Option Strict On

Standardmäßig erlaubt Visual Basic 9 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. Dies ermöglicht zwar eine bequeme und schnelle Programmierung, kann jedoch in größeren Projekten zu schwer auffindbaren Fehlern führen.

In Visual Basic 9 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.

In Visual Basic 9 wurde die Typenbehandlung noch um einiges vereinfacht. Die IDE macht nun Vorschläge, falls ein Datentyp in Gefahr ist, 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 korrekt durchgeführt werden. Bei größeren Zahlen, mit denen noch dazu umfangreiche Berechnungen angestellt würden, ist die Gefahr eines 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.

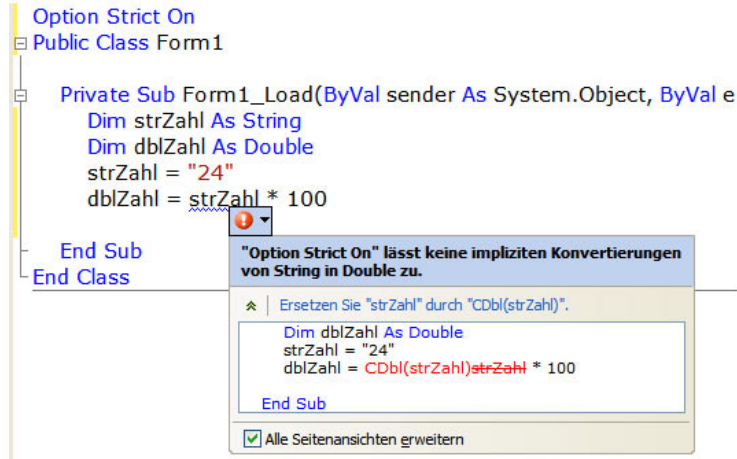
Wie Sie sehen, unterbreitet Ihnen Visual Studio gleich einen Vorschlag, wie Sie das Problem lösen können. Mit einem Klick wird `strZahl` zu `Cdbl(strZahl)` und Sie können mit der Arbeit an Ihrem Programm fortfahren.

**Tipp**

`Option Strict On` können Sie in Visual Studio 2008 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`.

Abbildung 3.2

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



### 3.1.10 Programmiernotation

Gerade für Fortgeschrittene stellt sich oft die Frage, wie Objekte, 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 wird es noch, 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 wurden der Gültigkeitsbereich der Variablen sowie der Variablentyp als Präfix geschrieben.

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

Mit .NET entfernte sich Microsoft aber von diesem Standard und fügte einige Neuerungen im Bereich der Framework-Programmierung hinzu.

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 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 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 Benachrichtigung gesendet. Diese Benachrichtigung kann daraufhin weiterverarbeitet werden (in Kapitel 5 erfahren Sie mehr dazu). 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 EventHandler
```

Ein Suffix sollte ebenso jedem Attribut mit angehängt werden.

```
Public Class ShowAllAttribute
```

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

```
Private vorname As String
```

## 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 vorherige 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 (auch in der Programmierung häufig 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.

#### Hinweis

In wurde der neue `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 9 stellt mit dem `IIf`-Operator nun endlich 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 darstellen. Ist der erste Aus-

druck 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()
```

### Hinweis

Die Funktionalität des IIF-Operators ist mit der Einführung von Visual Basic 9 verändert worden. Erst ab dieser Version handelt es sich tatsächlich um einen richtigen ternären Operator, da nur noch das Argument ausgewertet wird, 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:

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

**Listing 3.9**  
Select Case-Anweisung

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.



## 3 Visual Basic 9 allgemein

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 Werteblocken

```
Select Case ZuPrüfendeVariable
  Case 1 To 5
    'Befeh1
  Case 6 To 20
    'Befeh1
  Case 21 To 50
    'Befeh1
  Case Else
    'Befeh1
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 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()

```

### 3.3.3 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.

```

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

```

**Listing 3.11**  
For Next-Schleife  
im Einsatz

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

```

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

```

### 3.3.4 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 9 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.

### 3 Visual Basic 9 allgemein

**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 nun – 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. Dies kann auch später durch `ReDim` festgelegt werden, falls die Anzahl der Elemente noch nicht feststeht. Beachten Sie jedoch, dass `ReDim` sehr viel Rechenzeit benötigt.

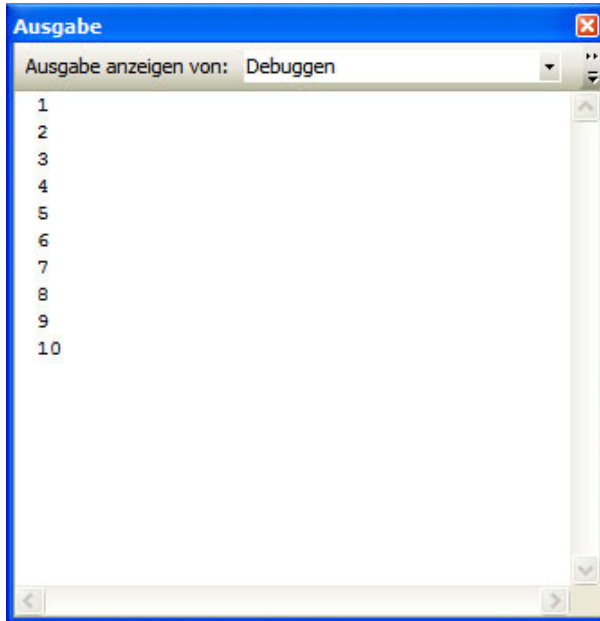
Im folgenden Listing 3.12 wird gezeigt, wie man ein Array mit Werten füllt und dieses dann in einer Konsole ausgeben kann.

#### 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 2008 aussehen wie in Abbildung 3.3.



**Abbildung 3.3**  
Ausgabe des Array-Beispiels in der Visual Studio-Konsole

### Hinweis

Die Ausgabe der Array-Werte erfolgt in einer Konsolenanwendung. Konsolenanwendungen lassen sich mit jeder Visual Studio 2008-Version erstellen und dienen zum Testen der Anwendung in einem Konsolenfenster ähnlich wie unter MS-DOS. Eine neue Konsolenanwendung legen Sie wie folgt an: Gehen Sie 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.

Zum anderen kann aber auch ein Array 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.

```
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)
```

**Listing 3.13**  
Ausgabe des Array-Objektes

### 3 Visual Basic 9 allgemein

**Listing 3.13 (Forts.)**  
Ausgabe des  
Array-Objektes

```
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 normale 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 Visual Basic 9 mit der `ReDim`-Anweisung Abhilfe.

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

```

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

```

Listing 3.15

Arrays mit  
ReDim erweitern

### 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.

```

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

```

Listing 3.16

Arrays sortieren

### 3 Visual Basic 9 allgemein

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 zeigt, wie Sie einem Array zunächst Werte von 1 bis 10 in aufsteigender Reihenfolge hinzufügen und diese dann umkehren.

**Listing 3.17**  
Arrays umkehren

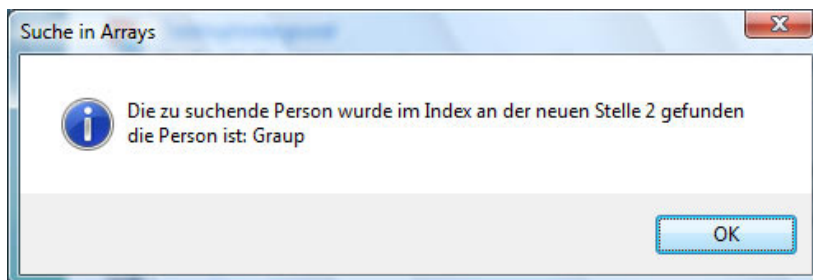
```
Public Sub Reverse()  
    Dim aWerte(9) As Integer  
    For i As Integer = 0 To 9  
        aWerte(i) = i + 1  
    Next i  
    Array.Reverse(aWerte)  
    For i As Integer = 0 To aWerte.Length - 1  
        Console.WriteLine(aWerte(i))  
    Next  
    Console.ReadLine()  
End Sub
```

### 3.4.5 Arrays durchsuchen

Nachdem das Array nun sortiert wurde, kann es auch durchsucht werden. Der Suchalgorithmus arbeitet auch bei großen Arrays noch 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 erfolgt die Ausgabe des neuen Index nach der Sortierung und des Namens der Person.

```
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 3.4.



**Listing 3.18**

Array nach bestimmten Wert durchsuchen

**Abbildung 3.4**

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 zur Durchführung von Berechnungen.
- 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 9 verwendet werden:

**Tabelle 3.6**  
Operatoren im  
Überblick

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

Die Darstellung der Grundrechenarten ist in Visual Basic 9 denkbar einfach. Hier einige Beispiele für die Verwendung von Grundrechenarten:

Beispiel für Addition:

```
db1Ausgabe = 10 + 3
```

Beispiel für Subtraktion:

```
db1Ausgabe = 10 - 3
```

Beispiel für Multiplikation:

```
db1Ausgabe = 10 * 3
```

Beispiel für Division:

```
db1Ausgabe = 10 / 3
```

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

```
db1Ausgabe = 10 ^ 3
```

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

### Achtung

Seien Sie 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
```

Eine Neuerung, die von vielen Visual Basic.NET-Programmierern gewünscht wurde, ist nun ebenfalls seit Visual Basic 2005 umgesetzt worden. Anstatt beim Hochzählen einer Variablen beispielsweise `i = i + 1` zu schreiben, können Sie nun den Shortcut `i += 1` verwenden. Dasselbe gilt für den Subtraktions-, Multiplikations- und Divisionsoperator.

Verknüpfungsooperatoren 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 Verknüpfungsooperator benötigt.

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

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

Visual Basic 9 kennt die in Tabelle 3.7 aufgelisteten logischen Operatoren.

Operator	Funktion
And	Und-Verknüpfung
Or	Oder-Verknüpfung
Not	exklusive Oder-Verknüpfung
Xor	Nicht-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

**Tabelle 3.7**  
Logische Operatoren

### 3 Visual Basic 9 allgemein

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 in False, andersherum ändert er den False-Wert in 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
```

## 3.6 Befehle

Komplette Programmanweisungen werden in Visual Basic 9 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 er 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.

## 3.7 Sonstige Sprachelemente

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

### 3.7.1 IsNot-Operator

Etwas umständlich war bislang die Abfrage, ob ein Objekt nicht `Nothing` oder etwas anderes nicht ist. Deswegen wurde jetzt der `IsNot`-Operator eingeführt, denn folgende Codezeile liest sich doch sicher intuitiver:

```
If obj IsNot Nothing Then
```

als

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

Das ist zwar nur eine kleine Änderung, macht aber Code durchaus lesbarer.

### 3.7.2 Weitere Schlüsselwörter

Seit Visual Basic 2005 stehen Ihnen drei neue Schlüsselwörter zur Verfügung, die wir uns hier noch kurz ansehen sollten.

#### 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 drei Jahre darum herumgekommen.

### 3 Visual Basic 9 allgemein

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 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 Collectors 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. Sie können jedoch 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 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 verarbeitet. Listing 3.20 zeigt eine Verwendung von `Continue`.

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

**Listing 3.20**  
Verwenden von  
`Continue`

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

## 3.8 Konsolenanwendungen erstellen

Visual Basic 9 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. Sie dienen als Grundlage für viele Visual Basic 9-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 denen 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.

### 3 Visual Basic 9 allgemein

#### 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 Konsolen-  
anwendung

```
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).

```
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()
```

#### Listing 3.24

Musik in der Konsole

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.