4 Variablen- und Objektverwaltung

Dieses Kapitel gibt eine Einführung in den Umgang mit Variablen (Deklaration, Zuweisung) und beschreibt einige grundlegende .NET-Datentypen, also beispielsweise *Integer*, *Double*, *Date* und *String*. Weitere Themen sind der Umgang mit Konstanten und Feldern. Das Kapitel endet mit einem Ausflug in die Interna der Variablen-, Objekt- und Speicherverwaltung, wobei sich dieser Abschnitt eher an fortgeschrittene Visual-Basic-Programmierer richtet.

Neu in Visual Basic 2008

- Variablen können ohne Typangaben deklariert werden, wenn ein Startwert angegeben wird (*Dim i = 2*, siehe Abschnitt 4.1). Das macht den Code kompakter, aber bisweilen auch unklarer: In manchen Fällen stehen mehrere geeignete Variablentypen zur Wahl, und es obliegt nun dem Compiler, wofür er sich entscheidet.

- Werttypen wie *Integer* oder *Date* können in den Formen *Dim i? As typ* oder *Dim i As typ?* als *Nullable*-Variablen deklariert werden. Das ermöglicht die zusätzliche Speicherung des Werts *Nothing* (siehe Abschnitt 4.5).

- Zur Verarbeitung von Feldern stehen eine Menge neuer generischer Methoden zur Verfügung. Beispielsweise ermittelt *feld.Sum()* die Summe aller Elemente.

  Die neuen Methoden sind allerdings nicht als Elemente der *Array*-Klasse definiert, sondern als *extension methods* in der Klasse *System. Linq.Enumerable*. (Was *extension methods* sind, verrät Abschnitt 6.7.) Die neuen Methoden werden allerdings nicht in diesem Kapitel vorgestellt, sondern in Abschnitt 7.2 zusammen mit anderen generischen Methoden zur Bearbeitung von Feldern und Aufzählungen.

> > > HINWEIS
>
> *Dieses Kapitel stellt wirklich nur eine Einführung dar! Detailinformationen finden Sie in den folgenden Kapiteln bzw. Abschnitten:*
>
> | | |
> |---|---|
> | *Lokale Variablen und Prozedurparameter:* | *Abschnitt 5.3* |
> | *Objektorientierte Programmierung:* | *Kapitel 6* |
> | *Eigene Klassen und Datenstrukturen definieren:* | *Abschnitt 6.5* |
> | *Gültigkeitsbereiche von Variablen in Klassen (scope):* | *Abschnitt 6.9* |
> | *Aufzählungen (***Collections***), generische Klassen und Methoden:* | *Kapitel 7* |
> | *Umgang mit Zahlen, Daten und Zeichenketten:* | *Kapitel 9* |
> | *Abfragen für Felder/Aufzählungen mit LINQ:* | *Kapitel 12* |

4.1 Einführung

Deklaration von Variablen

Variablen müssen vor ihrer ersten Verwendung mit ***Dim*** deklariert werden. Dieses Kommando kennt unglaublich viele Syntaxvarianten: Beispielsweise können Variablen innerhalb von Klassen auch mit *Private*, *Protected*, *Friend* oder *Public* deklariert werden. In diesem Kapitel beschränke ich mich aber auf *Dim*, alle anderen Varianten lernen Sie in Kapitel 6 kennen.

Im einfachsten Fall deklariert ***Dim x, y*** die Variablen *x* und *y*. Visual Basic verwendet dabei den Standarddatentyp *Object*. In derartigen Variablen können Sie alle erdenklichen Daten (Zahlen, Zeichenketten etc.) speichern, ohne dass Sie sich über den Datentyp Sorgen machen müssen. Die Variablendeklaration ohne Typenangabe ist nur zulässig, wenn *Option Strict Off* gilt. Das ist standardmäßig der Fall (siehe auch Abschnitt 4.6).

Dim x, y ***As Integer*** deklariert die beiden Variablen explizit als *Integer*-Variablen (32 Bit mit Vorzeichen). Der Vorteil gegenüber der Deklaration ohne die Angabe eines expliziten Datentyps besteht darin, dass Sie nun in *x* und *y* nicht versehentlich andere Daten speichern können. Außerdem bestehen aufgrund der expliziten Typangabe bessere Möglichkeiten zur Syntaxkontrolle, der resultierende Code wird effizienter, und der Platzbedarf im Speicher ist ein wenig geringer.

Das folgende Miniprogramm deklariert die beiden Variablen *x* und *y*, weist ihnen Werte zu und zeigt diese Werte dann in einer Hinweisbox an.

```
Module Module1
  Sub Main()
    Dim x, y As Integer
    x = 3
    y = x + 1
    MsgBox("x=" & x & "  y=" & y)
  End Sub
End Module
```

Standardmäßig müssen in Visual Basic Variablen vor ihrer ersten Verwendung deklariert werden. Es ist üblich, alle Variablendeklarationen am Beginn einer Prozedur (hier also am Beginn von *Main*) durchzuführen, das ist aber keine Bedingung.

Variablendeklaration mit Kennzeichnungszeichen

Für die wichtigsten Variablentypen gibt es spezielle Kennzeichner, die eine Kurzschreibweise bei der Deklaration ermöglichen: So können Sie statt *Dim x As Integer* auch die Kurzschreibweise *Dim x%* verwenden. (Ausführlichere Informationen über diese und weitere Variablentypen folgen im nächsten Abschnitt.)

Kennzeichner	Bezeichnung	Platzbedarf	Zahlenbereich
%	*Integer*	4 Byte	-2.147.483.648 bis 2.147.483.647
&	*Long*	8 Byte	-2^{63} bis $2^{63}-1$
@	*Decimal*	12 Byte	±9,99E27 mit 28 Stellen
!	*Single*	4 Byte	±3,4E38 mit 8 Stellen
#	*Double*	8 Byte	±1,8E308 mit 16 Stellen
$	*String*	10 + 2*n Byte	bis zu 2.147.483.647 Unicode-Zeichen

Die beiden Deklarationsformen können allerdings nicht nach Belieben gemischt werden. Insbesondere dürfen die Kurzschreibweisen nur allein oder nach *As*-Anweisungen angegeben werden, nicht aber davor.

```
Dim x%, y&                   'ok
Dim b1, b2 As Byte, x%, y&   'ok
Dim x%, y&, b1, b2 As Byte   'Syntaxfehler!
```

> > > HINWEIS

Die Visual-Basic-Dokumentation rät von der Verwendung von Kennzeichnern bei der Deklaration von Variablen ab. Derartige Kürzel vermindern die Lesbarkeit von Code. Allerdings sparen die Kürzel oft (gerade bei der Deklaration von Parametern) eine Menge Platz, was der Übersichtlichkeit sehr zugute kommt. Letztlich ist es eine persönliche Entscheidung (oder eine des Entwicklerteams), ob diese Kurzschreibweise angewandt wird oder nicht.

Variablennamen

Variablennamen müssen mit einem Buchstaben oder einem Unterstrich beginnen. Die weiteren Zeichen dürfen auch Zahlen enthalten. Es sind auch Sonderzeichen der jeweiligen Sprache erlaubt. Da Programmcode aber standardmäßig mit nur einem Byte pro Zeichen in Dateien gespeichert wird (ANSI), kann die Verwendung von Zeichen außerhalb des US-ASCII-Zeichensatzes Probleme verursachen. Es empfiehlt sich daher, auf die deutschen Zeichen äöü und ß in Variablennamen zu verzichten.

Informationen über die maximale Länge von Zeichenketten habe ich nicht gefunden, Experimente haben aber ergeben, dass diese größer als 256 Zeichen sein darf.

Variablennamen sollten nicht mit den Namen von Visual-Basic-Schlüsselwörtern übereinstimmen. Wenn dies unvermeidlich ist, können Sie den Variablennamen in eckige Klammern stellen. Beispielsweise ist *Dim [Next] As Integer* zulässig, obwohl *Next* ein Schlüsselwort zur Formulierung von Schleifen ist.

Groß- und Kleinschreibung

Grundsätzlich spielt in Visual Basic die Groß- und Kleinschreibung von Variablen keine Rolle. *variable* und *varIABle* und *Variable* sind also gleichwertig. Der Codeeditor passt die Schreibweise aller Variablen an die Schreibweise an, die bei der Deklaration verwendet wurde.

* * * TIPP

Wenn Sie die Schreibweise einer Variable in der Dim*-Anweisung verändern, ändert der Codeeditor nicht sofort alle anderen Zeilen, in denen die Variable vorkommt. Wenn Sie das möchten, markieren Sie einfach den gesamten Text mit* Strg+A *und klicken dann* ⭾ *an. Damit wird nicht nur der gesamte Code richtig eingerückt, sondern auch die Schreibweise aller Variablen synchronisiert.*

Wenn Sie eine Variable umbenennen möchten (also nicht nur die Groß- und Kleinschreibung ändern), klicken Sie die Variable mit der rechten Maustaste an und führen UMBENENNEN *aus. Damit erhält die Variable im gesamten Code (Deklaration und weitere Anwendung) einen neuen Namen.*

Wertzuweisung (Initialisierung) bei der Deklaration

Variablen können unmittelbar bei der Deklaration initialisiert werden. Die erforderliche Syntax sieht so aus:

```
Dim i1 As Integer = 1
Dim i2 As Integer = 2, i3 As Integer = 3
Dim i4% = 4, i5% = 5
```

Manche Datentypen und alle gewöhnlichen Klassen (Details folgen im nächsten Abschnitt) sehen zur Initialisierung von Variablen den *New*-Operator vor:

```
Dim d As New Date(2008, 12, 31)      'd1 enthält das Datum 31.12.2008
Dim s1 As New String("a", 3)         's1 enthält "aaa"
```

> > > HINWEIS

Was passiert, wenn Sie die Zuweisung a=b ausführen? Wird in a eine Kopie von b gespeichert oder ein Verweis auf b? Bevor diese Frage beantwortet werden kann, müssen Sie zunächst Objektvariablen kennenlernen. Die Antwort folgt dann in Abschnitt 4.4.

Deklaration ohne Typangabe (implizite Typisierung)

Eine Neuerung in VB2008 besteht darin, dass Sie Variablen ohne explizite Typangabe deklarieren können. Die einzige Voraussetzung besteht darin, dass Sie zur Initialisierung der Variablen einen Startwert angeben, anhand dessen der VB-Compiler einen geeigneten Datentyp auswählen kann. Diese Technik wird auch *implicit typing* oder *local type inference* genannt bzw. in der deutschsprachigen Dokumentation fallweise *Typrückschluss*. Ich bleibe in diesem Buch beim Begriff *implizite Typisierung*.

```
Dim i = 2                                'entspricht Dim i As Integer = 2
Dim s = "abc"                            'entspricht Dim s As String = "abc"
Dim d = 1.2                              'entspricht Dim d As Double = 1.2
Dim pf = PixelFormats.Bgr32              'entspricht Dim pf As PixelFormat = ...
Dim st As New IO.StreamReader("dateiname")  'entspricht Dim st As IO.StreamReader = ...
```

> > > HINWEIS

*Bei **Dim i = 0** entscheidet sich der VB-Compiler für den Datentyp **Integer**. Wenn Sie eine **Double**-Variable wünschen, müssen Sie **Dim i = 0.0** angeben.*

*Bei ganzen Zahlen im 32-Bit-Integer-Zahlenraum verwendet der VB-Compiler immer den Datentyp **Integer** (32 Bit), nie **Long** (64 Bit). Ob das angesichts der immer stärkeren Verbreitung von 64-Bit-CPUs eine gute Entscheidung ist, wird sich in ein paar Jahren herausstellen. Ich sehe auf jeden Fall schon alle möglichen Kompatibilitätsprobleme vor mir, wenn der Compiler von VB2012 (oder wie auch immer die übernächste VB-Version heißen wird) sich plötzlich für **Long** entscheiden wird ... Wenn Sie schon jetzt **Long**-Variablen wünschen, müssen Sie die Variable mit **Dim i = 3&** oder klarer mit **Dim i As Long = 3** deklarieren.*

Die implizite Typisierung ist auch in der Deklaration von Feldern sowie für Schleifenvariablen zulässig:

```
Dim ar() = New Integer() {1, 2, 3}  'entspricht Dim ar As Integer = ...
For Each var In ar                  'entspricht For Each var As Integer In ar
```

! ! ! ACHTUNG

Ebenfalls zulässig ist die folgende Schreibweise:

Dim numbers() = {1, 2, 3}

*Man würde annehmen, dass **numbers** damit ebenfalls als **Integer**-Feld deklariert wird. Aber weit gefehlt! Der VB-Compiler entscheidet sich hier aus unerfindlichen Gründen für ein **Object**-Feld. Ein **Integer**-Feld liefern nur die folgenden alternativen Anweisungen. (Beachten Sie auch den Einsatz der Klammern zur Kennzeichnung eines Felds.)*

Dim numbers = New Integer() {1, 2, 3}
Dim numbers() = New Integer {1, 2, 3}
Dim numbers1() As Integer = {1, 2, 3}
Dim numbers1() As Integer() = {1, 2, 3}

Im folgenden Beispiel ist *contact* eine Klasse oder eine Struktur mit den Elementen *name*, *email* und *tel*. Da die Feldelemente neue Objekte dieser Klasse sind, ist für das Feld *contact* keine Typangabe erforderlich. Beachten Sie auch die neue Syntax mit *With* zur Initialisierung der Elemente!

```
Dim contacts() = { _
  New contact With {.name = "a", .email = "a@x", .tel = "123"}, _
  New contact With {.name = "b", .email = "b@x", .tel = "124"}}
```

Die implizite Typisierung ist nur für lokale Variablen erlaubt, nicht aber für Eigenschaften, Funktionen, Klassenelemente etc. Es liegt auf der Hand, dass der Compiler oft einen gewissen Entscheidungsspielraum hat: Je nach Anwendung wäre es im Einführungsbeispiel vielleicht zweckmäßiger, *i* als *Byte*- oder eine *Long*-Variable zu definieren.

Es ist nicht auszuschließen, dass sich künftige VB-Versionen bei derselben Deklaration für einen anderen Variablentyp entscheiden. Persönlich bin ich kein Freund solcher Ungenauigkeiten, die schon in der Vergangenheit viel Ärger verursacht haben. Ein weiterer Nachteil besteht darin, dass gedruckter Code (also ohne den *IntelliSense*-Komfort der Entwicklungsumgebung) schwerer nachvollziehbar wird. Andererseits hat die implizite Typisierung auch Vorteile: In vielen Fällen wird der Code dadurch kompakter, leichter lesbar und weniger redundant.

Wie dem auch sei, es ist Ihre persönliche Entscheidung (oder die Ihres Projektleiters), ob Sie implizite Typisierung einsetzen oder nicht. Bei Bedarf können Sie die implizite Typisierung durch die Option *Option Infer Off* verbieten – wahlweise für das gesamte Projekt in den Compiler-Einstellungen der Projekteigenschaften oder für das aktuelle Modul bzw. die aktuelle Klasse durch die Angabe der Option am Beginn des Moduls bzw. der Klasse.

Beachten Sie aber, dass die implizite Typisierung für manche LINQ-Anwendungen unverzichtbar ist. Standardmäßig gilt für neue VB2008-Projekte *Option Infer On*. Alte Projekte, die von VB2005 oder einer früheren VB-Version konvertiert wurden, verwenden dagegen *Option Infer Off*!

Standardwerte von nicht initialisierten Variablen

Es ist schlechter Programmierstil, eine Variable zu nutzen, bevor Sie ihr explizit einen Wert zugewiesen haben. Bei *String*- und Objektvariablen warnt der Compiler vor solchen Situationen. Die folgende Tabelle gibt an, welche Startwerte die nicht initialisierten Variablen haben, die auftreten, wenn Sie sich über die Warnung des Compilers hinwegsetzen.

Variablentyp	Startwert	Bemerkungen
numerische Variablen	0	
Boolean-Variablen	*False*	
String-Variablen	*Nothing*	Vorsicht: Nicht initialisierte *String*-Variablen enthalten tatsächlich *Nothing*, auch wenn man oft den Eindruck gewinnt, sie enthielten "". Der Grund für diese Missverständnisse ist die Visual-Basic-*Runtime*: Diese interpretiert nicht initialisierte *String*-Variablen wie eine leere Zeichenkette ""! *Len(s)* liefert daher *0*. Der Vergleich *s=""* liefert (eigentlich inkorrekt) *True*. *s Is Nothing* und *IsNothing(s)* liefern korrekt *True*. Verwenden Sie die nicht initialisierte *String*-Variable dagegen im Kontext von .NET-Methoden (z.B. *s.Length*), tritt ein Fehler auf.
Char-Variablen	0	Vorsicht: Die Entwicklungsumgebung, d.h. der Editor, das Kommandofenster und die Überwachungsfenster, zeigen bei nicht initialisierten *Char*-Variablen *Nothing* an. Das ist falsch! *System.Convert.ToInt16(c)* beweist, dass *c* wirklich ein Zeichen mit dem Code 0 enthält. *IsNothing(c)* funktioniert korrekt und liefert *False*. *c Is Nothing* kann für *Char*-Variablen nicht ausgewertet werden.
Date-Variablen	*#12:00:00 AM#*	Vorsicht: Diese Zeitangabe entspricht bei deutscher Formatierung 00:00:00 (und nicht etwa 12 Uhr mittags)! Intern ist das Datum 1.1. des Jahres 0001 gespeichert. *d.Ticks* enthält 0.
Object-Variablen	*Nothing*	Objektvariablen werden im nächsten Abschnitt beschrieben.

4.2 Variablentypen

Dieser Abschnitt gibt einen Überblick über die elementaren Variablentypen (Datentypen) von Visual Basic. Genau genommen handelt es sich dabei um .NET-Klassen wie *System. Boolean*, *System.Byte* etc., die unter Visual Basic aber zum Teil unter anderen Namen verwendet werden können. Beispielsweise lautet die Visual-Basic-Bezeichnung für *System. Int16* einfach *Short*.

> > > HINWEIS

Dieser Abschnitt stellt die elementaren Datentypen nur kurz vor. Eine ausführliche Beschreibung der vielen Methoden, die es zur Bearbeitung, Formatierung und Konvertierung von Zahlen, Daten und Zeichenketten gibt, finden Sie in Kapitel 9.

Ganze Zahlen (Byte, Short, Integer, Long)

Die folgende Tabelle fasst die in Visual Basic vorgesehenen Datentypen zur Speicherung ganzer Zahlen zusammen. Die in der ersten Spalte angegebenen Zeichen können als Kurzschreibweise zur Kennzeichnung des Datentyps verwendet werden. Die beiden folgenden Deklarationen sind deswegen gleichwertig:

```
Dim i As Integer
Dim i%
```

Die Angaben für den Platzbedarf beziehen sich auf die eigentlichen Daten. Je nach Anwendung (z.B. wenn die Daten in einer als *Object* deklarierten Variable gespeichert werden) kann der tatsächliche Platzbedarf deutlich größer sein (siehe auch Abschnitt 4.10).

Kürzel	Bezeichnung	.NET-Bezeichnung	Platzbedarf	Zahlenbereich
	Boolean	*System.Boolean*	1 Byte	*True* oder *False*
	Byte	*System.Byte*	1 Byte	0 bis 255
	SByte	*System.SByte*	1 Byte	-128 bis 127
	Short	*System.Int16*	2 Byte	-32.768 bis 32.767
	UShort	*System.UInt16*	2 Byte	0 bis 65.535
%	*Integer*	*System.Int32*	4 Byte	-2.147.483.648 bis 2.147.483.647
	UInteger	*System.UInt32*	4 Byte	0 bis 4.294.967.295
&	*Long*	*System.Int64*	8 Byte	-2^{63} bis $2^{63}-1$
	ULong	*System.Int64*	8 Byte	0 bis $2^{64}-1$

Fließ- und Festkommazahlen (Single, Double, Decimal)

Der Standarddatentyp für Fließkommazahlen und Fließkommaberechnungen ist *Double*. Dieser Datentyp kann auch intern am effizientesten verarbeitet werden. *Single* sollte nur dann eingesetzt werden, wenn der Platzbedarf eine wichtige Rolle spielt (etwa bei riesigen Feldern).

* * * TIPP

Die Division x = 1.0 / 0.0 löst (anders als bei ganzen Zahlen) keinen Fehler aus! Stattdessen ist das Resultat einer derartigen Division der Wert **Double.NegativeInfinity** *oder* **Double.PositiveInfinity**. *Einige weitere Informationen zu diesem interessanten Aspekt von* **Single**- *und* **Double**-*Zahlen finden Sie in Abschnitt 9.1.*

Der Datentyp *Decimal* eignet sich insbesondere zur Speicherung von Werten, bei denen keine Rundungsfehler auftreten dürfen. *Decimal*-Variablen sind daher besonders gut geeignet, wenn Geldbeträge verarbeitet werden sollen. Die Genauigkeit beträgt 28 Stellen, wobei die Position des Kommas variabel ist. (Wenn eine Zahl 10 Stellen vor dem Komma beansprucht, stehen für den Nachkommaanteil noch 18 Stellen zur Verfügung.)

Decimal-Variablen sind aber auch mit Nachteilen verbunden: Rechenoperationen werden viel langsamer als mit *Double*-Variablen ausgeführt, der Speicherbedarf ist größer, und anders als bei *Single* und *Double* ist keine Exponentialdarstellung zulässig. Deswegen ist der zulässige Zahlenbereich viel kleiner. Der bis VB6 zur Verfügung stehende Datentyp *Currency* wird von Visual Basic nicht mehr unterstützt (verwenden Sie *Decimal*!).

Kürzel	Bezeichnung	.NET-Bezeichnung	Platzbedarf	Zahlenbereich
@	*Decimal*	*System.Decimal*	12 Byte	±9,99E27 mit 28 Stellen
#	*Double*	*System.Double*	8 Byte	±1,8E308 mit 16 Stellen
!	*Single*	*System.Single*	4 Byte	±3,4E38 mit 8 Stellen

Datum und Uhrzeit (Date)

Anders als in VB6, wo Daten und Zeiten als *Double*-Wert gespeichert werden, erfolgt die interne Repräsentierung nun durch einen *Long*-Wert. Dieser Wert gibt die Anzahl sogenannter *Ticks* zu je 100 ns (Nanosekunden) an, die seit dem 1.1.0001 00:00 vergangen sind. Mit *datevar.Ticks* können Sie diesen internen Wert auslesen. Im Programmcode werden Daten und Zeiten am einfachsten in der Form *#mm/dd/yyyy#* bzw. *#hh:mm:ss#* dargestellt. *d = #12/31/2008 23:59:59#* bezeichnet also den Zeitpunkt eine Sekunde vor Neujahr 2009.

System.DateTime sieht eigentlich einen Zeitbereich vom 1.1.0001 bis zum 31.12.9999 vor. Die Visual-Basic-Dokumentation spricht hingegen vom 1.1.100 als kleinstmöglichem Datum. Diese Aussage resultiert nicht aus den internen Möglichkeiten von *DateTime*, sondern aus der Art und Weise, wie Visual Basic mit zweistelligen Jahreszahlen umgeht. Wenn Sie ein zweistelliges Datum zuweisen (z.B. *datevar=#12/31/15#*), wird die Jahreszahl automatisch in 1930 bis 2029 umgewandelt (beim gewählten Beispiel also in 2015). Ein Sonderfall ist die Zuweisung einer Uhrzeit ohne Datumsangabe: In diesem Fall verwendet auch VB als Datum den 1.1.0001.

Kürzel	Bezeichnung	.NET-Bezeichnung	Platzbedarf	Zeitbereich
	Date	*System.DateTime*	8 Byte	1.1.0001 00:00:00 bis 31.12.9999 23:59:59

Zeichenketten (String)

Zeichenketten werden intern im Unicode-Format gespeichert (UTF16, also mit zwei Byte pro Zeichen). Das ermöglicht die Speicherung fast aller ausländischer Sonderzeichen.

Allerdings speichert die Entwicklungsumgebung standardmäßig Visual-Basic-Code in ANSI-Dateien (ein Byte pro Zeichen, Codierung entsprechend der Codeseite 1252). Wenn Sie im Code Unicode-Zeichen verwenden möchten, die sich außerhalb dieser Codeseite befinden, müssen Sie den Programmcode im Unicode-Format abspeichern. Dazu führen Sie DATEI | ERWEITERTE SPEICHEROPTIONEN aus und wählen einen der zur Auswahl stehenden Unicodes aus.

Eine Besonderheit bei Zeichenketten besteht darin, dass es sich hierbei um unveränderliche (*immutable*) Objekte handelt. Wenn Sie *x = x + "abc"* ausführen, wird eine neue Zeichenkette gebildet und die alte Zeichenkette verworfen. (Die alte Zeichenkette wird automatisch aus dem Speicher entfernt.) Das gilt selbst dann, wenn sich die Länge der Zeichenkette nicht ändert oder wenn sich die Zeichenkette verkürzt.

Leere (nicht initialisierte) *String*-Variablen haben den Wert *Nothing*. Einige Visual-Basic-Funktionen interpretieren diesen Wert so, als würde die Variable eine leere Zeichenkette enthalten. Leere *Char*-Variablen haben den Wert *Chr(0)*.

Kürzel	Bezeichnung	.NET-Bezeichnung	Platzbedarf	Inhalt
	Char	*System.Char*	2 Byte	ein Unicode-Zeichen
$	*String*	*System.String*	ca. 10 + 2**n* Byte	bis zu 2.147.483.647 Unicode-Zeichen

Objekte

Wenn Sie mit *Dim* eine Variable deklarieren, ohne explizit einen Typ anzugeben, verwendet Visual Basic *Object* als Standardtyp. In *Object*-Variablen können beliebige Daten gespeichert werden – Zahlen, Zeichenketten, Daten und Zeiten sowie Objekte aller .NET-Klassen.

Object ist daher der allgemeingültigste Variablentyp von Visual Basic. Wenn Sie sich keine Gedanken über den richtigen Datentyp machen möchten, können Sie immer *Object*-Variablen verwenden. Visual Basic kümmert sich nun selbst darum, intern den richtigen Variablentyp einzusetzen. Der Preis dieser Bequemlichkeit ist aber hoch: Ihr Programm wird ineffizient und fehleranfällig (d.h., Sie werden viele Fehler erst bemerken, wenn Sie das Programm tatsächlich ausführen).

Obwohl *Object* auf den ersten Blick ähnliche Eigenschaften wie der aus VB6 vertraute Variablentyp *Variant* hat, ist er intern ganz anders realisiert. Es handelt sich dabei um die Überklasse (*System.-Object*), von der alle Variablentypen und .NET-Klassen abgeleitet sind. Die *Object*-Variable besteht im Wesentlichen aus einem Zeiger (*pointer*) auf die tatsächlichen Daten. Insofern beträgt der Speicherbedarf für eine noch nicht initialisierte *Object*-Variable nur vier Byte. Sobald die Variable tatsächlich benutzt wird, kommt aber noch der Speicherbedarf für die tatsächlichen Daten dazu.

Bezeichnung	.NET-Bezeichnung	Platzbedarf
Object	*System.Object*	4 bzw. 8 Byte für den Zeiger bei 32-Bit bzw. 64-Bit-Programmen, plus *n* Byte für die Daten

In diesem Buch werden Sie kaum auf *Object*-Variablen stoßen. Stattdessen sind Objektvariablen fast immer exakt deklariert. Die folgende Anweisung deklariert *dir* beispielsweise als Variable, in der ein Objekt der Klasse *System.IO.DirectoryInfo* gespeichert werden kann:

```
Dim dir As IO.DirectoryInfo
```

Weitere .NET-Datentypen

Neben den in diesem Abschnitt vorgestellten Datentypen kennt das .NET-Framework eine Reihe weiterer Datentypen. Im Objektbrowser finden Sie diese Datentypen in der *System*-Klasse der mscorlib-Bibliothek, die allen Visual-Basic-Programmen zur Verfügung steht. Die folgende Tabelle nennt lediglich die zwei interessantesten Datentypen.

.NET-Datentypen, die von Visual Basic nicht direkt unterstützt werden	
System.Guid	128-Bit-Integerzahlen zur Speicherung von *globally unique identifier*
System.TimeSpan	Zeitspannen (relative Zeitangaben, Beispiele siehe Abschnitt 9.4)

CLS-Datentypen

Wenn Sie mit Visual Basic Steuerelemente oder Bibliotheken entwickeln, die kompatibel zu anderen .NET-Programmiersprachen sind, sollten Sie für die Schnittstellen ausschließlich CLS-Datentypen einsetzen. CLS steht für *Common Language Specification* und beschreibt einen Standard für .NET-Bibliotheken. Fast alle in diesem Abschnitt beschriebenen Datentypen sind CLS-konform. Ausnahmen sind *SByte*, *UShort*, *UInteger*, *ULong*, *GUID* und *TimeSpan*.

CLS-konforme Datentypen	
Boolean, Byte, Short, Integer, Long	ganze Zahlen
Decimal, Double, Single	Fließkommazahlen
Date	Datum und Uhrzeit
String, Char	Zeichenketten

Sie müssen Ihre Bibliothek explizit mit dem Attribut *<assembly:CLSCompliant(true)>* als CLS-konform markieren. Weitere Informationen finden Sie in der Hilfe zu *CLSCompliantAttribute*.

4.3 Werttypen (ValueType) versus Referenztypen

Was sind Objekte?

Ein Kapitel zur Variablenverwaltung wäre ohne die Berücksichtigung von Objekten unvollständig. Allerdings werden Objekte und die Grundzüge objektorientierter Programmierung erst in den folgenden Kapiteln ausführlich beschrieben. Daher ist hier ein Vorgriff unvermeidbar. Wenn Ihnen die Beschreibung der Nomenklatur an dieser Stelle zu schnell geht, muss ich Sie also auf die nachfolgenden Kapitel vertrösten.

Klasse (Typ): Eine Klasse beschreibt die Eigenschaften eines Objekts. Die Klasse ist gewissermaßen der Bauplan für ein Objekt. In der .NET-Klassenbibliothek sind mehrere Tausend Klassen definiert, die bei der Organisation und Verwaltung aller möglichen Dinge helfen:

Es gibt Klassen zur Beschreibung der Eigenschaften einer Datei (z.B. *System.IO.FileInfo*), Klassen zur Darstellung eines Fensters (z.B. *System.Windows.Forms.Form*), Klassen für alle Steuerelemente, die im Fenster angezeigt werden (z.B. *System.Windows.Forms.Button*), Klassen zur Erzeugung von Zufallszahlen (*System.Random*) etc. Darüber hinaus können Sie selbst eigene Klassen definieren.

Statt des Begriffs Klasse wird manchmal auch der Begriff Typ verwendet, vor allem in der Zusammensetzung Werttyp bzw. Referenztyp. Gemeint sind damit Klassen mit bestimmten Merkmalen – siehe unten. Nicht initialisierte Objekte enthalten den Zustand ***Nothing***.

Vererbung: Klassen können ihre Eigenschaften und Methoden an andere Klassen gewissermaßen vererben. Das bedeutet, dass mehrere von einer Basisklasse abgeleitete Klassen dieselben Grundeigenschaften und -methoden haben. Vererbung ist ein wichtiges Werkzeug bei der Programmierung eigener Klassen.

Objekte: Objekte sind konkrete Realisierungen von Klassen. In Variablen speichern Sie daher Objekte, nicht Klassen. Wenn die folgende Zeile ausgeführt wird, dann enthält die Variable *myRandomObject* ein Objekt der Klasse *System.Random*:

```
Dim myRandomObject As New System.Random()
```

Werttypen (ValueType) versus Referenztypen

Viele Programmiersprachen differenzieren zwischen gewöhnlichen Variablen (z.B. für Integerzahlen) und Objektvariablen. Nicht so Visual Basic: Hier ist jede Variable eine Objektvariable!

Wenn hier und in vielen anderen Büchern dennoch manchmal zwischen *Variablen* und *Objektvariablen* differenziert wird, dann bezieht sich diese Unterscheidung auf den Typ der zugrunde liegenden Klasse. Die .NET-Bibliothek teilt nämlich alle Klassen in zwei ganz wesentliche Typen ein:

- **Werttypen:** Zu dieser Gruppe zählen unter anderem alle elementaren Datentypen (z.B. *Integer*, *Double* etc.) mit der Ausnahme von *String*. Das entscheidende interne Merkmal besteht darin, dass sie von der Klasse ***ValueType*** abgeleitet sind. (Daraus resultiert auch die Bezeichnung *ValueType*-Klassen bzw. *ValueType*-Objekte.)

  Neben den elementaren Datentypen gibt es eine ganze Reihe von Klassen und Datenstrukturen in der .NET-Bibliothek, die ebenfalls von *ValueType* abgeleitet sind. Zumeist handelt es sich dabei um eher kleine Klassen (klein hinsichtlich des Speicherbedarfs, aber auch klein hinsichtlich der angebotenen Funktionen). Zwei Beispiele sind *System.Drawing.Rectangle* und *System.Drawing.Color*. Auch die in Visual Basic definierten Strukturen (*Structure ... End Structure*) und Aufzählungen (*Enum*) werden von *ValueType* abgeleitet.

- **Referenztypen:** Zu dieser Gruppe zählen alle Klassen, die nicht von *ValueType* abgeleitet sind. Die Bezeichnung *Referenztypen* (im Englischen *reference types* oder *reference classes*) resultiert daraus, dass Objekte dieser Klassen als Referenz (als Zeiger) in Variablen gespeichert bzw. an Prozeduren oder Methoden übergeben werden.

Zu den Referenztypen zählt die Mehrheit aller Klassen der .NET-Bibliothek, z.B. *System.Array*, *System.IO.FileInfo*, *System.Drawing.Bitmap* und *System.Windows.Forms.Button*, um willkürlich vier Beispiele herauszugreifen. *ValueType*-Klassen sind so gesehen nur eine Ausnahme.

Beachten Sie, dass auch Felder (die intern durch die Klasse *System.Array* verwaltet werden) zu den Referenztypen zählen – selbst dann, wenn deren Elemente oft Werttypen sind (z.B. Integerzahlen).

Die Motivation für die Trennung zwischen Wert- und Referenztypen lautet kurz und einfach: Effizienz. Damit der objektorientierte Ansatz von .NET konsequent verfolgt werden kann, müssen alle Daten Objekte sein. Durch die Behandlung als Objekte entsteht aber ein hoher Overhead, der bei einfachen Daten (z.B. bei Integerzahlen) in keinem Verhältnis zum Nutzen stehen würde. Deswegen muss es für den Compiler einen Weg geben, mit manchen Objekten (eben mit den *ValueType*-Objekten) so umzugehen, als wären es nur einfache Werte.

Objekte, die von Wert- bzw. von Referenztypen abgeleitet sind, unterscheiden sich in ihrer Verwendung ganz erheblich: bei Variablenzuweisungen, bei der Übergabe als Parameter an eine Prozedur oder Methode, bei der internen Speicherverwaltung etc. In diesem und in den folgenden Kapiteln wird immer wieder auf diesen Unterschied hingewiesen. Daher ist es wichtig, dass Sie zwischen Wert- und Referenztypen differenzieren können!

Wie können Sie aber feststellen, welchem Typ eine bestimmte Klasse angehört? Im laufenden Programm werten Sie dazu einfach *obj.GetType().IsValueType* aus. Während der Programmentwicklung offenbart der Objektbrowser (ANSICHT|OBJEKTBROWSER) den Klassentyp. Dort suchen Sie die Klasse und sehen sich deren Definition an (rechts unten im Objektbrowser). Werttypen sind als *Structure* deklariert, Referenztypen als *Class*. Abbildung 4.1 zeigt, dass die Klasse *System.Windows.Media.Color* ein Werttyp ist.

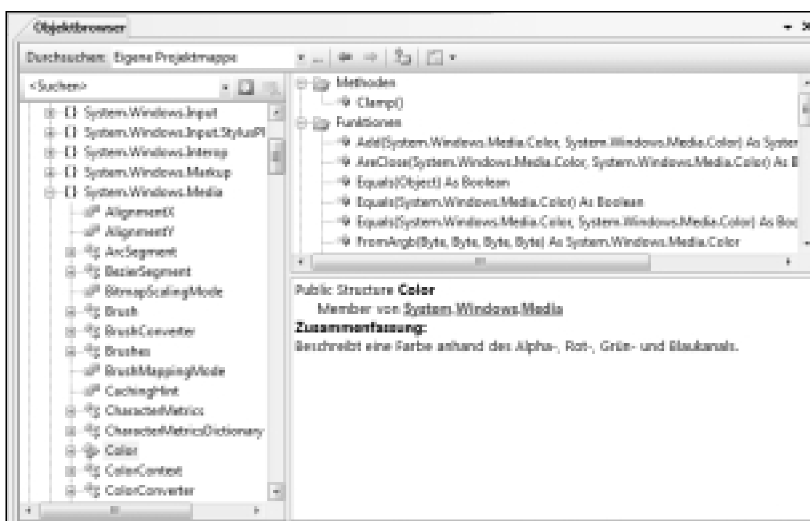

Abbildung 4.1: Die Klasse System.Windows.Media.Color ist eine ValueType-Klasse.

> > > HINWEIS
>
> *Wert- und Referenztypen verhalten sich vollkommen unterschiedlich, wenn sie mit* ***ByVal*** *an eine Prozedur übergeben werden! Details dazu finden Sie in Abschnitt 5.3!*

Deklaration von Objektvariablen

Wenn Sie eine Variable deklarieren, die auf ein bestimmtes Objekt verweisen soll, ändert sich an der Syntax nichts. Statt des Datentyps geben Sie jetzt den Klassennamen ein:

```
Dim myRandomObject As System.Random
```

Die Variable *myRandomObject* ist damit allerdings noch leer! Bevor Sie Methoden der *System.Random*-Klasse nutzen können, müssen Sie das Objekt erst erzeugen:

```
myRandomObject = New System.Random()
```

Im Regelfall werden Sie diese beiden Anweisungen mit ***Dim As New*** in einer einzigen Zeile vereinen:

```
Dim myRandomObject As New System.Random()
```

Noch kompakter wird die Zeile, wenn Sie auf die Angabe von *System* verzichten (was im Regelfall möglich ist – mehr dazu in Abschnitt 6.2):

```
Dim myRandomObject As New Random()
```

Bei vielen Klassentypen können an die *New*-Methode Parameter übergeben werden. Die folgende Anweisung erzeugt ein Objekt der Klasse *System.IO.DirectoryInfo*. Das Objekt wird gleich mit dem Pfad des aktuellen Verzeichnisses initialisiert.

```
Dim dir As IO.DirectoryInfo = New IO.DirectoryInfo(".")
```

> > > HINWEIS
>
> *Bei Werttypen wie* **Integer** *oder* **String** *können Variablen sofort nach der Deklaration verwendet werden. Das Schlüsselwort* **New** *ist zwar erlaubt und manchmal zur Initialisierung auch nützlich, es ist aber nicht erforderlich. Hintergrundinformationen über die Unterschiede zwischen Wert- und Referenztypen bei der Speicherverwaltung folgen in Abschnitt 4.10.*

Objektvariablen ohne Typangabe

Wenn Sie im Voraus nicht wissen, wie Sie eine Objektvariable verwenden werden, können Sie die Variable sehr allgemein als *Object* deklarieren. Sie können diese Variable dann im weiteren Verlauf nach Belieben verwenden. Durch *myObject=3* wird in *myObject* ein *Integer*-Wert gespeichert. Durch *New System.Random* wird ein Objekt des Typs *System. Random* gespeichert etc.

```
Dim myObject As Object
myObject = 3                     'myObject verweist jetzt auf eine Integer-Variable
myObject = New System.Random()  'myObject verweist jetzt auf ein Objekt der
                                 ' Klasse System.Random
```

Diese Vorgehensweise ist allerdings mit Nachteilen verbunden: Erstens weiß die Entwicklungsumgebung nicht, wie Sie die Variable verwenden werden, und kann daher bei der Codeeingabe nicht die für ein Objekt relevanten Methoden und Eigenschaften vorschlagen. Zweitens weiß auch der Compiler nicht, wie Sie die Variable verwenden, und kann daher nur eine eingeschränkte Syntaxkontrolle durchführen. Drittens kann es passieren, dass Sie der Variable versehentlich einen Wert zuweisen und so unbeabsichtigt den Objekttyp verändern. (Dabei kommt es zu keinem Fehler. Probleme kann es aber später geben, wenn Sie Objekteigenschaften oder -methoden einsetzen, die es für den neuen Objekttyp gar nicht gibt.) Und zu guter Letzt ist der Code etwas langsamer – aber dieses Argument ist wahrscheinlich das unwichtigste.

Fazit: Geben Sie bei der Deklaration nach Möglichkeit exakt den Daten- oder Objekttyp an! Sie ersparen sich damit eine oft langwierige Fehlersuche.

Verschiedene Wege, um den Typ einer Variablen zu bestimmen, werden in Abschnitt 4.11 beschrieben. Am vielseitigsten ist dabei die Methode *GetType*.

4.4 Variablenzuweisungen

Was passiert, wenn Sie *a* = *b* ausführen? Diese Frage klingt trivial, aber wenn es wirklich so trivial wäre, gäbe es dazu natürlich keinen eigenen Abschnitt. Das Ergebnis einer Variablenzuweisung hängt nämlich von der Art der Variablen ab. Variablen zur Speicherung von Werttypen verhalten sich anders als solche für Referenztypen.

Dieser Abschnitt setzt voraus, dass *a* und *b* für denselben Daten- bzw. Objekttyp deklariert wurden. Wenn das nicht der Fall ist, kommt es bei der Zuweisung zu einer automatischen Konvertierung der Daten. Diese gelingt allerdings nur in Sonderfällen (z.B. wenn *a* eine *Double*- und *b* eine *Integer*-Variable ist). Wenn eine verlustfreie Typumwandlung dagegen nicht möglich ist, kommt es zu einer Fehlermeldung. Mehr Informationen zum Thema der automatischen und manuellen Typkonvertierung erhalten Sie in den Abschnitten 4.12 und 9.7.

Werttypen (ValueType-Variablen)

Bei Werttypen, d.h. bei den meisten elementaren Datentypen (*Integer*, *Double*, *Date* etc.), wird durch *a* = *b* eine *Kopie* der Daten erstellt und zugewiesen.

```
Dim a As Integer = 1
Dim b As Integer = 2
a = b  'jetzt ist a=2 und b=2
b = 3  'jetzt ist a=2 und b=3
```

Referenztypen (herkömmliche Objektvariablen)

Bei Referenztypen (deren Klasse nicht von *ValueType* abgeleitet ist) wird durch *a = b* ein Link (eine Referenz) auf das Objekt zugewiesen, also keine Kopie! Im folgenden Beispiel wird in *a* und *b* jeweils ein *Button*-Objekt gespeichert. Derartige Objekte dienen normalerweise zur Darstellung von Buttons in einem Fenster. Die Zuweisung *a = b* bewirkt hier, dass nun beide Variablen auf denselben Button verweisen. Durch die Veränderung der Eigenschaft *b.Text* ändert sich nun auch *a.Text* (weil ja *a* und *b* auf dasselbe Objekt verweisen)!

```
Dim a As New Windows.Forms.Button()
Dim b As New Windows.Forms.Button()
a.Text = "a"
b.Text = "b"
a = b          'a und b zeigen jetzt auf denselben Button (.Text="b")
b.Text = "x"   'damit wird .Text für a und für b verändert!
```

Wenn Sie bei gewöhnlichen Objekten eine Kopie der Daten benötigen, müssen Sie *a = b.**Clone()*** ausführen. Allerdings steht die Methode *Clone* nicht für alle Klassen zur Verfügung. Wenn es *Clone* nicht gibt, besteht keine Möglichkeit, ein Objekt zu kopieren. (Die *Button*-Klasse sieht kein *Clone* vor.)

Vielleicht fragen Sie sich, was mit dem ursprünglich für *a* erzeugten Button passiert: Es gibt nun ja keine Variable mehr, die darauf verweist. Deswegen wird das Objekt nach einiger Zeit automatisch aus dem Speicher entfernt. Dieser Prozess wird *garbage collection* genannt und ist in Abschnitt 4.10 beschrieben.

String-Variablen

String-Variablen zur Speicherung von Zeichenketten stellen einen Sonderfall dar. Zwar zählt *String* zu den elementaren Datentypen, es handelt sich aber intern dennoch um einen Referenztyp. (Die *String*-Klasse ist nicht von *ValueType* abgeleitet.) Und trotzdem verhalten sich *String*-Variablen wie Werttypen!

Der Grund: *String*-Objekte gelten als unveränderlich (*immutable*). Das bedeutet, dass bei jeder Veränderung einer Zeichenkette ein vollkommen neues *String*-Objekt erzeugt wird. Aus diesem Grund betrifft eine Änderung immer nur eine Variable (auch wenn vorher zwei Variablen auf dasselbe *String*-Objekt verwiesen haben).

```
Dim a As String = "a"
Dim b As String = "b"
a = b          'a und b verweisen nun auf "b"
b = "x"        'a verweist weiterhin auf "b", b verweist auf das neue String-Objekt "x"
```

Geradezu verblüffend ist das Verhalten, wenn Sie nun auch *a* die Zeichenkette *"x"* zuweisen: Dann verweisen *a* und *b* wieder auf *dasselbe String*-Objekt (d.h., der Objektvergleich *a Is b* liefert *True*).

```
a = "x"        'a und b verweisen nun auf dasselbe String-Objekt "x"
```

Dieses Verhalten gilt allerdings nicht immer. Wenn Sie zuerst *a="12"* und dann *b="1"* und *b+="2"* ausführen, dann enthalten zwar *a* und *b* dieselbe Zeichenkette *"12"*, aber intern verweisen die beiden Variablen auf unterschiedliche Objekte (weil der Compiler hier nicht erkennen konnte, dass die Ergebnisse der Zuweisungen gleichartige Objekte sein würden).

4.5 Nullable für Werttypen

Variablen für Referenztypen (*Object*, *String* sowie alle von *Object* abgeleiteten Klassen) enthalten *Nothing*, wenn sie nicht initialisiert sind. Bei Werttypen besteht diese Möglichkeit dagegen nicht. Nicht initialisierte Variablen erhalten automatisch einen Standardwert. Bei einer *Integer*-Variable mit dem Wert 0 lässt sich daher nicht sagen, ob der Wert 0 das Ergebnis einer Zuweisung ist oder ob die Variable bisher noch nie verwendet worden ist. Diese Differenzierung ist vor allem in Datenbankanwendungen interessant. Dort kommt es oft vor, dass in den Spalten einer Tabelle der SQL-Wert *NULL* zulässig ist. So kann beispielsweise unterschieden werden, ob der Lagerbestand für ein Produkt 0 ist oder noch nicht erfasst wurde (*NULL*).

Bereits seit VB2005 können Variablen für Werttypen als ***Nullable*** definiert werden. Damit enthalten diese Variablen standardmäßig *Nothing*. In Zuweisungen kann der Wert *Nothing* auch explizit gespeichert werden. *i* ist also eine *Integer*-Variable, die außer ganzen Zahlen auch den Wert *Nothing* enthalten kann. (*Nothing* in *Visual Basic* entspricht *Null* in C# und anderen Sprachen. Die Bezeichnung *Nullable* ist in Visual Basic irreführend – eher müsste es *Nothingable* heißen. Aber das klingt noch schlimmer ...)

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

Neu in VB2008 sind zwei kompaktere Syntaxvarianten, die dieselbe Wirkung wie die obige Zeile haben:

```
Dim i? As Integer
Dim i As Integer?
```

Sie können *Nullable* übrigens nicht für *String*-Variablen anwenden. Der Grund: *String* ist ein Referenztyp. *Nothing* ist daher von Grund auf ein zulässiger Wert, eine Deklaration mit *Nullable* ist daher überflüssig.

Nullable-Eigenschaften

*var.**Value*** liefert den Wert der Variable, wenn diese initialisiert ist. Enthält *var* dagegen *Nothing*, wird eine *InvalidOperationException* ausgelöst. *var.Value* ermöglicht auch den Zugriff auf die Eigenschaften des zugrunde liegenden Datentyps (z.B. *d.Value.Year* für *Dim d? As Date*).

*var.**HasValue*** liefert *True*, wenn die Variable initialisiert ist, bzw. *False*, wenn *var Nothing* enthält.

*var.**GetValueOrDefault**(standardwert)* liefert den Wert der Variable, wenn diese initialisiert ist, bzw. den Standardwert des Variablentyps, wenn *var Nothing* enthält.

Umgang mit Nullable-Variablen

Viele Einschränkungen, die in VB2005 für den Umgang mit *Nullable*-Variablen galten, sind in VB2008 behoben: Sie können nun Berechnungen direkt durchführen (*i = i + 1* statt *i = i.Value + 1* in VB2005), *Nullable*-Variablen als Schleifenvariablen verwenden etc. Einige Einschränkungen gibt es aber weiterhin:

- Der Operator *Is* kann nur in der Form *If var Is Nothing* verwendet werden. Der Vergleich zweier *Nullable*-Variablen in der Form *If var1 Is var2* ist nicht möglich.
- Der Zugriff auf Eigenschaften ist umständlich: Wenn *Dim d? As Date* gilt, dann können Sie die *Date*-Eigenschaften wie *Year*, *Month* etc. nicht direkt verwenden. Die Eigenschaften sind nun via *d.**Value*** zugänglich. Üblicherweise werden Sie dabei auch einen Test durchführen, ob *d* überhaupt Daten enthält. Die folgende Anweisung speichert in *y* das Jahr oder den Wert 0, wenn *d* leer ist:

  ```
  y = If(d.HasValue, d.Value.Year, 0)
  ```

- Die automatische Typumwandlung bei Zuweisungen (eine Visual-Basic-Spezialität) funktioniert nur mit Einschränkungen. Beispielsweise liefert *var = row!spaltenname* einen Fehler, obwohl *var* eine *Nullable-Integer*-Variable ist und *row!spaltenname* eine Integerzahl enthält! Abhilfe schafft nur der Einsatz von Casting-Funktionen wie *CInt* oder *CDate*.
- ADO.NET ist nicht in der Lage, den SQL-Wert *NULL* bzw. den Wert *DBNull.Value* unmittelbar *Nothing* zuzuordnen. Wenn die Spalte einer Tabelle also *NULL* enthalten kann, müssen Sie die Zuweisung an eine *Nullable*-Variable wie folgt durchführen:

  ```
  Dim var As Nullable(Of Integer)
  If row!spaltenname Is DBNull.Value Then
    var = Nothing
  Else
    var = CInt(row!spaltenname)
  End If
  ```

  Ein äquivalentes Problem tritt auch auf, wenn Sie einer Spalte eines *DataRow*-Objekts eine *Nullable*-Variable zuweisen möchten. Die korrekte Vorgehensweise ist umständlich:

  ```
  If var.HasValue Then
    row!spaltenname = var.Value
  Else
    row!spaltenname = DBNull.Value
  End If
  ```

  Das ist eine besonders absurde Einschränkung, weil *Nullable* ja gerade eingeführt wurde, um Datenbankprogrammierern das Leben zu erleichtern. Weitere Tipps zum Umgang mit *NULL*-Werten gibt Abschnitt 27.8.

4.6 Option Explicit, Strict und Infer

Option Explicit (On), *Option Strict (Off)* und *Option Infer* (*On*, neu in VB2008) sind im Dialogblatt KOMPILIEREN der Projekteigenschaften (*My Project*) voreingestellt. Sie können diese Einstellung wahlweise im Projekteigenschaftsdialog ändern oder am Beginn des Codes eine *Option*-Anweisung hinzufügen. Die zweite Variante hat den Vorteil, dass Sie die Option für jede einzelne Codedatei individuell einstellen können.

```
Option Explicit On|Off
Option Strict On|Off
Option Infer On|Off
```

Die Standardeinstellung dieser drei Optionen sowie von *Option Compare* (siehe Abschnitt 10.3) für *neue* Projekte legen Sie mit EXTRAS | OPTIONEN | PROJEKTE UND PROJEKTMAPPEN | VB-STANDARD fest.

Option Explicit

Option Explicit On (die Standardeinstellung) bewirkt, dass alle Variablen vor ihrer Verwendung durch *Dim* deklariert werden müssen. Das verhindert, dass Tippfehler in Variablennamen unbemerkt bleiben. Es gibt keinen vernünftigen Grund, diese Einstellung zu ändern.

Option Strict

Diese Option steuert, ob der Visual-Basic-Compiler diverse Codevarianten zulässt, z.B. die automatische Umwandlung zwischen verschiedenen Datentypen (implizite Typkonvertierung), *late binding* sowie die Deklaration von Variablen ohne Datentypangabe (implizite Typdeklaration). *late binding* bedeutet, dass bei *Object*-Variablen Methoden und Eigenschaften genutzt werden können. Ob diese Methoden und Eigenschaften tatsächlich zur Verfügung stehen, hängt vom Inhalt der Objektvariablen ab. Das kann erst bei der Programmausführung überprüft werden – daher also *late binding*.

Option Strict kann wahlweise im Code für die jeweilige Datei mit *On* oder *Off* eingestellt werden oder für das gesamte Projekt im Dialogblatt KOMPILIEREN der Projekteigenschaften. In diesem Dialog können Sie auch für einzelne Sonderfälle Warnungen aktivieren. Standardmäßig gilt für neue Projekte *Option Strict Off*, wobei alle damit verbundenen Warnungen deaktiviert sind.

> > > HINWEIS

Persönlich bin ich kein Fan dieses saloppen Umgangs mit Datentypen. In der Vergangenheit habe ich meine Projekte zumeist mit **Option Strict Off** *entwickelt. In VB2008 ist das aber zunehmend unmöglich, insbesondere, wenn Sie mit LINQ arbeiten oder Beispielcode aus der Hilfe bzw. aus dem Internet ausprobieren möchten.*

Die Eleganz diverser neuer Sprachmerkmale in VB2008, insbesondere von LINQ, besteht eben gerade darin, den Code so knapp wie möglich zu formulieren. Für exakte Typangaben ist da kein Platz mehr. Diese Unsitte greift übrigens auch unter C# um sich. Das hat immerhin den Vorteil, dass C#-Anhänger nicht mehr mit erhobenem Zeigefinger auf die ach so schlampigen VB-Programmierer losgehen können.

Im Zusammenhang mit der Variablenverwaltung sind nur die implizite Konvertierung sowie die implizite Typangabe (*Dim* ohne Datentyp) interessant. Für die Standardeinstellung *Option Strict Off* gilt, dass beides ohne Warnungen erlaubt ist. Wenn Sie dagegen Wert auf exakten Code legen, sollten Sie *Option Strict* aktivieren. Sie müssen nun bei allen Umwandlungen, die das Risiko eines Datenverlusts in sich bergen, explizit eine Konvertierungsfunktion angeben:

```
Dim i As Integer, d As Double = 1.25
i = CInt(d)
```

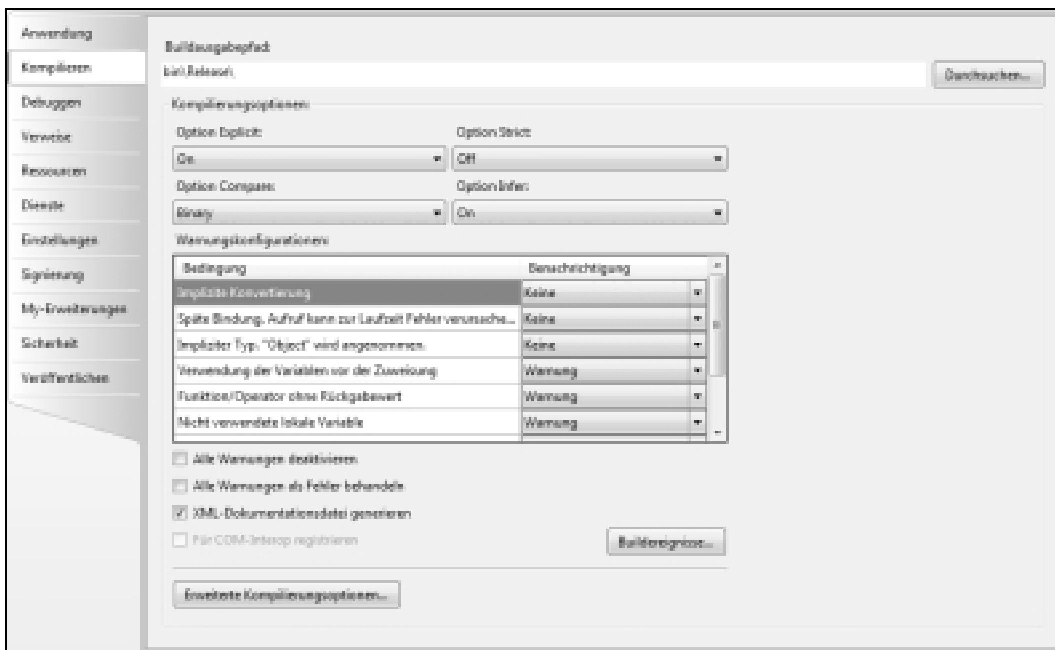

Abbildung 4.2: Projekteigenschaften

Option-Strict-Beispiel

Die folgenden Zeilen sehen so aus, als würden sie zur aktuellen Zeit in *d1* eine Stunde hinzuaddieren. Tatsächlich kommt es bei der Durchführung der Addition aber zu einem Fehler. (In VB6 funktionierte dieser Code tatsächlich. In aktuellen Visual-Basic-Versionen sind derartige Additionen aber nicht zulässig.)

```
Dim d1, d2, d3 As Date
d1 = Now
d2 = #1:00:00 AM#
d3 = d1 + d2       'Fehler in Visual Basic!
```

Was hat nun dieser Code mit *Option Strict* zu tun? Visual Basic betrachtet + hier als einen Operator, um zwei Zeichenketten zu verknüpfen! *d1* und *d2* werden entsprechend der Ländereinstellung auto-

matisch in Zeichenketten umgewandelt und aneinandergefügt. Der Fehler tritt erst auf, weil die ebenfalls automatische Rückkonvertierung der Zeichenkette in ein Datum für die Zuweisung an *d3* scheitert. Die aus *d1+d2* resultierende Zeichenkette lautet *"29.11.2008 16:23:2101:00:00"*, wenn der Code am 29.11.2008 um 16:23 mit der deutschen Ländereinstellung ausgeführt wurde.

Wenn Sie *Option Strict On* verwenden, erkennt die Entwicklungsumgebung sofort, dass hier Probleme auftreten werden. Ohne *Option Strict* tritt der Fehler dagegen erst bei der tatsächlichen Ausführung des Codes auf.

Option Infer

Diese Option ist neu in VB2008. Sie steuert, ob eine Deklaration von Variablen ohne explizite Typangabe in der Form *Dim i = 2* zulässig ist (siehe auch Abschnitt 4.1). Die Standardeinstellung ist *On* für neue Projekte, aber *Off* für Projekte, die mit früheren VB-Versionen erstellt worden sind. Dass auch *Option Infer* nicht ohne Risiken ist, zeigt die folgende Anweisung:

```
Dim ar() = New {1, 2, 3}   'liefert ein Object-Feld, kein Integer-Feld!
```

4.7 Konstanten

Konstanten selbst definieren

Mit *Const* können Sie Konstanten deklarieren. Die Syntax von *Const* entspricht weitgehend der von *Dim*. Der Unterschied besteht darin, dass die so deklarierten Konstanten unveränderlich sind.

```
Const const1 As Integer = 3, const2 As Double = 4.56
```

Eine merkwürdige Eigenheit von selbst definierten Konstanten besteht darin, dass sie beim Testen von Programmen in der Entwicklungsumgebung nicht sichtbar sind. Konstanten können weder im Befehlsfenster verwendet noch in Überwachungsfenstern angezeigt werden.

Vordefinierte VB-Konstanten

In Visual Basic stehen zahllose vordefinierte Konstanten zur Verfügung. Die folgende Tabelle ist alles andere als komplett. Sie soll lediglich als erste Orientierungshilfe dienen. Eine Beschreibung aller Konstanten in *Microsoft.VisualBasic* finden Sie, wenn Sie in der Hilfe nach *Konstanten Enumerationen* suchen.

Viele Definitionen sind doppelgleisig: Sowohl *ControlChars.Tab* als auch *vbTab* enthalten den Code 9 (ein Tabulatorzeichen), und sowohl *MsgBoxResult.Abort* als auch *vbAbort* enthalten den Wert 3 etc. Es ist eine Geschmacksfrage, welche Konstanten Sie verwenden.

Der Vorteil der *vbXxx*-Konstanten besteht darin, dass diese in VB-Programmen unmittelbar verwendet werden können. Bei allen anderen Konstanten muss jeweils die dazugehörende Klasse ange-

geben werden (*ControlChars* für *Char*-Konstanten, *MsgBoxResult* und *MsgBoxStyle* für Konstanten zum Aufruf und zur Auswertung von *MsgBox* etc.).

Konstante	Verwendung	Definiert in
True = -1, False = 0	enthalten Wahrheitswerte.	Visual-Basic-Sprachdefinition
Nothing	gibt an, dass die Variable nicht initialisiert (also leer) ist.	Visual-Basic-Sprachdefinition
vbXxx (z.B. *vbCrLF*, *vbTab*, *vbYes* etc.)	erhöhen die Kompatibilität mit VB6.	*Microsoft.VisualBasic.Constants* (Teil der Visual-Basic-*Runtime*)
Back, *Cr*, *CrLf*, *FormFeed*, *Lf*, *NewLine*, *NullChar*, *Quote*, *Tab*, *VerticalTab*	dienen zur Zusammensetzung von Zeichenketten (siehe Abschnitt 10.1).	*Microsoft.VisualBasic.ControlChars* (Teil der Visual-Basic-*Runtime*)

.NET-Konstanten

Neben den VB-spezifischen Konstanten gibt es in den .NET-Klassenbibliotheken unzählige weitere Konstanten. Diese sind sehr oft als sogenannte *Enum*-Aufzählungen realisiert (siehe den folgenden Abschnitt), was ihre Anwendung erleichtert. Falls Sie keine entsprechenden Importe definiert haben, müssen Sie den Konstanten ihren Namensraum voranstellen, also z.B. *IO.FileAttributes.Compressed* für die Konstante *Compressed*.

4.8 Enum-Aufzählungen

Der Begriff Aufzählung ist zweideutig. Er wird einerseits dazu verwendet, um die hier beschriebenen *Enum*-Konstrukte zu benennen. (Das sind Gruppen von Konstanten.) Andererseits meint Aufzählung oft auch ein Objekt des *System.Collections*-Namensraums. Damit können Sie Listen verwalten, in die Sie jederzeit Elemente einfügen können und aus denen Sie diese Elemente auch wieder löschen können. Diese Art von Aufzählungen wird in Kapitel 7 beschrieben.

Syntax und Anwendung

Mit der Anweisung ***Enum*** *name* – ***End Enum*** können Sie eine ganze Gruppe von Konstanten definieren. Die folgenden Zeilen demonstrieren die Syntax:

```
Enum myColors As Integer
  Red       'automatisch Wert 0
  Green     'automatisch Wert 1
  Blue = 10 'Wert 10
  Yellow    'automatisch Wert 11
End Enum
```

mycolors ist nun ein neuer Datentyp. Sie können also eine Variable vom Typ *mycolors* deklarieren:

```
Dim col As myColors
```

Wenn Sie mit *Option Strict* arbeiten, können Sie dieser Variable ausschließlich die im *Enum*-Block definierten Konstanten zuweisen:

```
col = myColors.Green   'OK
col = 0                'nicht erlaubt, falls Option Strict On
```

Auch bei Vergleichen können Sie ausschließlich die *Enum*-Konstanten verwenden:

```
If col = myColors.Red Then
  ...
End If
```

Der Vorteil eines *Enum*-Blocks besteht also darin, dass die so definierten Konstanten nur im Kontext von entsprechenden *Enum*-Variablen verwendet werden können. Das erleichtert die Codeeingabe (der Editor schlägt automatisch die zulässigen Konstanten vor) und vermindert die Gefahr, versehentlich (für die jeweilige Variable oder Datenstruktur) ungeeignete Konstanten zu verwenden.

Enum-Konstrukte können innerhalb von Modulen, Klassen und Strukturen sowie auf äußerster Ebene im Code (also außerhalb anderer Konstrukte) definiert werden. Es ist aber nicht möglich, eine *Enum*-Aufzählung innerhalb einer Prozedur zu definieren.

Zahlenwert eines Enum-Elements ermitteln: Nach Möglichkeit sollten Sie im Programmcode beim Umgang mit *Enum*-Aufzählungen ausschließlich die definierten Konstanten verwenden (für Zuweisungen, Vergleiche etc.). Sollten Sie aus irgendeinem Grund dennoch den Zahlenwert benötigen, können Sie diesen jederzeit mit den VB-Konvertierungsfunktionen ermitteln (also etwa *CInt(col)* oder *CInt(myColors.Red)*).

***Enum*-Datentyp:** Als Datentyp für die Konstanten kommen *Byte*, *Short*, *Integer* oder *Long* in Frage. Laut Dokumentation muss der Datentyp in der ersten Zeile des *Enum*-Blocks angegeben werden, wenn *Option Strict* verwendet wird. Tatsächlich ist die Angabe des Datentyps aber anscheinend immer optional, wobei *Integer* als Standarddatentyp gilt.

Automatische Werte: Sie können jedem *Enum*-Element einen beliebigen (ganzzahligen) Wert zuweisen. Wenn Sie das nicht tun, verwendet Visual Basic standardmäßig 0 für das erste Element, 1 für das zweite etc. Nach den expliziten Wertzuweisungen setzt Visual Basic automatisch mit *n+1* fort.

Enum-Elemente mit gleichen Werten: Es ist erlaubt, dass zwei *Enum*-Elemente denselben Wert enthalten (auch wenn das selten sinnvoll ist). Passen Sie auf, dass das nicht unbeabsichtigt passiert!

```
Enum choices As Integer
  good = 10     'Wert 10
  medium        'automatisch Wert 11
  bad           'automatisch Wert 12
  horrible = 12 'ebenfalls 12!
End Enum
```

Enum-Kombinationen (Flags)

Manchmal wollen Sie in *Enum*-Variablen nicht einen einzelnen Zustand, sondern eine Kombination von Zuständen speichern. Ein typisches .NET-Beispiel hierfür ist *System.IO.FileAttributes*: Damit werden mehrere Attribute von Dateien gleichzeitig ausgedrückt (z.B. *Hidden* und *ReadOnly*).

Wenn Sie selbst eine derartige *Enum*-Klasse deklarieren möchten, müssen Sie der Definition *<Flags()>* voranstellen. Die eckigen Klammern bedeuten, dass es sich beim Inhalt um ein sogenanntes Attribut handelt, das der nachfolgenden Deklaration zusätzliche Eigenschaften verleiht. Hier wird als Attribut *Flags* verwendet. Intern bewirkt das, dass die *Enum*-Klasse zusätzliche Eigenschaften der Klasse *System.FlagsAttribute* erhält. (Was Attribute eigentlich sind, wird in Abschnitt 6.17 erklärt.)

Bei der Definition der Konstanten müssen Sie darauf achten, dass jede Konstante eine Zweierpotenz ist (1, 2, 4, 8, 16, 32 etc.). Damit stellen Sie sicher, dass auch jede Kombination von Konstanten eindeutig ist. Sie können bei der Zuweisung der Konstanten auch hexadezimale Werte verwenden (*&H1*, *&H2*, *&H4*, *&H8*, *&H10*, *&H20* etc.).

Das folgende Beispiel zeigt die Definition der *Enum*-Klasse *myPrivileges*, mit der Zugriffsrechte verwaltet werden (z.B. für ein Dateisystem, eine Datenbank etc.). Es ist damit beispielsweise möglich, jemandem den Lesezugriff sowie die Ausführung von Programmen zu erlauben, aber Veränderungen zu verbieten.

```
<Flags()> Enum myPrivileges As Integer
  ReadAccess = 1
  WriteAccess = 2
  Execute = 4
  Delete = 8
End Enum
```

Bei der Verwendung von *myPrivileges*-Variablen können Sie die verschiedenen Zustände mit *Or* verknüpfen. (Beachten Sie, dass die mathematisch ebenfalls korrekte Verknüpfung der Konstanten durch + bei *Option Strict* nicht zulässig ist!)

```
Dim priv As myPrivileges
priv = myPrivileges.ReadAccess Or myPrivileges.Execute
```

ToString liefert nun eine Zeichenkette, die die Kombination aller *Enum*-Konstanten ausdrückt. (Ohne das *Flags*-Attribut bei der *Enum*-Deklaration würde das nicht funktionieren!)

```
s = priv.ToString        ' s = "ReadAccess, Execute"
```

Wenn Sie testen möchten, ob eine Variable eine bestimmte Konstante enthält, können Sie nicht mehr einfach einen Vergleich mit = durchführen (weil dieser Vergleich natürlich *False* liefert, wenn die Variable eine Kombination von Konstanten enthält). Stattdessen müssen Sie den Vergleich wie das folgende Beispiel mit *And* formulieren:

```
If (priv And myPrivileges.Execute) <> 0 Then
    ...
End If
```

Beachten Sie, dass die Klammern und der Ausdruck <> *0* erforderlich sind, wenn Sie mit *Option Strict On* arbeiten. (*a And b* liefert einen *Integer*-Ausdruck, *If* erwartet aber einen *Boolean*-Ausdruck!)

Interna der System.Enum-Klasse

Intern sind *Enum*-Konstrukte von der .NET-Klasse *System.Enum* abgeleitet. Das bedeutet insbesondere, dass Sie alle Eigenschaften und Methoden von *System.Enum* auf *Enum*-Konstanten und -Variablen anwenden können. Dieser Abschnitt geht auf einige Interna von *Enum*-Konstrukten ein. Dabei wird das Wissen bzw. Verständnis von Grundtechniken der objektorientierten Programmierung vorausgesetzt, das erst im weiteren Verlauf dieses Buchs vermittelt wird. Dieser Abschnitt richtet sich daher an Leser, die schon etwas Erfahrung mit Visual Basic haben.

Namen (Zeichenketten) von Enum-Elementen ermitteln: Wenn Sie bei einem gegebenen *Enum*-Wert den Namen des entsprechenden Elements wissen möchten, verwenden Sie einfach die Methode ***ToString***:

```
Dim s As String, col As myColors
col = myColors.Green
s = col.ToString   's = "Green"
```

Alle Namen einer Enum-Aufzählung ermitteln: Die folgende Schleife verwendet ***GetNames***, um alle in *myColors* definierten Namen anzuzeigen (also *"Red"*, *"Green"*, *"Blue"* und *"Yellow"*). Beachten Sie, dass die Kurzschreibweise *Enum.GetNames* nicht zulässig ist (obwohl *System* sonst fast immer weggelassen werden kann), weil *Enum* ein Visual-Basic-Schlüsselwort ist. *GetNames* erwartet als Parameter ein *Type*-Objekt, das den Datentyp beschreibt. Ein derartiges Objekt wird hier mit *col.**GetType**()* erzeugt.

```
Dim all() As String, s As String, col As myColors
all = System.Enum.GetNames(col.GetType())
For Each s In all
  Console.WriteLine(s)
Next
```

Enum-Element aus Zeichenkette erzeugen: Die Methode ***Parse*** wertet die angegebene Zeichenkette aus und liefert als Ergebnis ein *Enum*-Objekt, das der Zeichenkette entspricht.

Die Definition von *Parse* gibt an, dass die Methode ein Objekt des Typs *Object* zurückgibt. Daher muss ***CType*** verwendet werden, um eine Umwandlung in ein *myColors*-Objekt durchzuführen. (*CType* wird in Abschnitt 4.12 beschrieben.) Die folgende Anweisung entspricht *col = myColors.Red*:

```
Dim col As myColors
col = CType(System.Enum.Parse(col.GetType(), "Red"), myColors)
```

Standardmäßig unterscheidet *Parse* zwischen Groß- und Kleinschreibung. Wenn Sie das nicht möchten, müssen Sie als dritten Parameter *True* übergeben:

```
col = CType(System.Enum.Parse(col.GetType(), "rEd", True), myColors)
```

Parse kommt auch mit *Enum*-Kombinationen zurecht. In der Zeichenkette müssen die einzelnen Namen durch Kommas getrennt werden. Die folgende Anweisung entspricht *priv = myPrivileges.-ReadAccess Or myPrivileges.WriteAccess*:

```
Dim priv As myPrivileges
priv = CType(System.Enum.Parse(priv.GetType(),"ReadAccess, WriteAccess"), myPrivileges)
```

Testen, ob ein Enum-Wert gültig ist: Wenn Sie wissen möchten, ob ein beliebiger Wert einer *Enum*-Konstante entspricht, können Sie dies mit ***IsDefined*** feststellen. Diese Methode liefert *True* oder *False*:

```
If System.Enum.IsDefined(col.GetType(), 17) Then ...
```

Beachten Sie, dass *IsDefined* für *Enum*-Kombinationen ungeeignet ist! *IsDefined(priv, 10)* würde eigentlich der Kombination *WriteAccess Or Delete* entsprechen, liefert aber *False*!

Syntaxzusammenfassung

Aufzählungen deklarieren und verwenden	

Enum aufz As Byte/Short/Integer/Long
  element1 [= wert1]
  element2 [= wert2]
  ...
End Enum
	deklariert eine Aufzählung. Den Elementen werden automatisch fortlaufende Zahlen zugewiesen, wenn Sie nicht explizit eigene Werte angeben.

<Flags()> Enum komb_aufz As ...
  element1 = 1
  element2 = 2
  element3 = 4
  ...
End Enum
	deklariert eine Aufzählung, deren Elemente kombiniert werden können. Die Elemente müssen dazu Zweierpotenzen enthalten.

Dim aufz_obj As aufz	deklariert *aufz_obj* als Variable der Aufzählung.

Eigenschaften und Methoden der Klasse *System.Enum*	
*System.Enum.**GetNames**(aufz.GetType())*	liefert ein Zeichenkettenfeld, das die Namen aller *Enum*-Konstanten enthält.
*System.Enum.**IsDefined**(aufz.GetType(), n)*	testet, ob *n* ein gültiger Wert einer *Enum*-Konstante von *aufz* ist.
*aufz_obj = CType(System.Enum.**Parse**(_* *col.GetType(), s), aufz)*	wertet die Zeichenkette *s* aus und liefert die entsprechende *Enum*-Konstante von *aufz*.

4.9 Felder

Felder kommen immer dann zum Einsatz, wenn Sie mehrere gleichartige Daten (z.B. Zeichenketten, Integerzahlen etc.) effizient verwalten möchten.

Intern sind alle Felder von der .NET-Klasse *System.Array* abgeleitet. Das bedeutet, dass Sie alle Eigenschaften und Methoden von *System.Array* zur Bearbeitung von Feldern anwenden können. Beachten Sie bitte, dass die Methoden von *System.Array* zum Teil direkt auf Feldvariablen angewendet werden können (z.B. *feld.Rank*), zum Teil aber das Feld als Parameter erwarten (z.B. *Array.-Reverse(feld)*).

> > > HINWEIS

Felder sind nicht die einzige Möglichkeit zur Verwaltung von Daten! Die .NET-Bibliothek bietet eine ganze Gruppe sogenannter **Collection***-Klassen, die für Spezialanwendungen effizienter sind als Felder. Damit können Sie beispielsweise assoziative Felder bilden (bei denen ein beliebiges Objekt und nicht eine Integerzahl als Index gilt), komfortabel Elemente einfügen und löschen etc. Einen Überblick über die wichtigsten* **Collection***-Klassen sowie Tipps zu ihrer Anwendung gibt Kapitel 7.*

Syntax und Anwendung

Felder werden in Visual Basic ganz ähnlich wie Variablen deklariert. Der einzige Unterschied besteht darin, dass an den Variablennamen oder an den Variablentyp ein Klammernpaar angehängt werden muss, um so zu kennzeichnen, dass es sich um ein Feld handelt.

```
Dim a() As Integer     'Elementzahl noch unbekannt
Dim a As Integer()     'gleichwertige Alternative
```

Im Regelfall geben Sie auch gleich die Anzahl der Elemente an. In diesem Fall muss die Anzahl im ersten Klammernpaar angegeben werden. Eine Besonderheit von Visual Basic besteht darin, dass *Dim b(3)* ein Feld mit *vier* Elementen deklariert: *a(0)*, *a(1)*, *a(2)* und *a(3)*. Der Zugriff auf einzelne Elemente erfolgt in der Form *feldname(indexnummer)*.

```
Dim b(3) As Integer   'eindimensionales Feld, vier Elemente
b(0) = 17
b(1) = 20
b(2) = b(0) + b(1)
b(3) = -7
```

Sie können Felder auch direkt bei der Deklarierung initialisieren. Bei dieser Syntaxvariante dürfen Sie allerdings keine Elementzahl angeben – Visual Basic entscheidet selbst, wie viele Elemente erforderlich sind. Beim folgenden Beispiel hat *c* drei Elemente: *c(0)*, *c(1)* und *c(2)*.

```
Dim c() As Integer = {7, 12, 39}
```

Ab VB2008 ist dank impliziter Typisierung (mit *Option Infer On*) auch die folgende Syntax erlaubt. (Die beiden Zeilen unterscheiden sich durch den Ort des Klammernpaars. Beachten Sie, dass sowohl

Dim c = New Integer { ... } als auch *Dim c() = New Integer() { ... }* unzulässig sind. Es muss genau ein Klammernpaar verwendet werden.)

```
Dim c() = New Integer {7, 12, 39}
Dim c = New Integer() {7, 12, 39}
```

! ! ! ACHTUNG

Die folgende Anweisung liefert überraschenderweise kein **Integer**-, *sondern ein* **Object**-*Feld!*

Dim c() = {7, 12, 39}

Wenn Sie ein Feld an eine Methode übergeben, können Sie das Feld auch dynamisch beim Aufruf der Methode übergeben. Die folgende Zeile führt die Methode *AddRange* für ein *ListBox*-Steuerelement aus. Diese Methode erwartet ein beliebiges Feld als Parameter.

```
ListBox1.Items.AddRange( New String() {"a", "b", "c"} )
```

In manchen Fällen kann es zweckmäßig sein, ein Feld vorerst leer zu initialisieren. Dazu weisen Sie der Funktion einfach nur *{}* zu. Stellen Sie sich beispielsweise eine Funktion vor, die ein *String*-Feld mit Namen als Ergebnis liefert. Wenn kein passender Name gefunden wird, soll ein leeres Feld zurückgegeben werden:

```
Public Function myfunction() As String()
  Dim result As String() = {}
  If no_data_found Then
    Return result  'oder: Return New String() {}
  Else
    ReDim result(n)
    result(..)=...
    Return result
  End If
End Function
```

Felder neu dimensionieren

Eine Besonderheit von Visual Basic besteht darin, dass Sie Felder mit ***ReDim*** nachträglich neu dimensionieren können. Wenn Sie dabei das optionale Schlüsselwort ***Preserve*** angeben, bleibt der Inhalt des bisherigen Felds erhalten. (Beachten Sie bitte, dass *ReDim* ein verhältnismäßig zeitaufwendiger Vorgang ist. Wenn Sie eine große Anzahl von Elementen verwalten, wobei die genaue Anzahl von vornherein nicht feststeht, sollten Sie entweder auf eine der in Kapitel 7 beschriebenen *Collection*-Klassen zurückgreifen oder die Feldanzahl nur in großen Schritten erhöhen.)

```
ReDim a(7)
a(5) = 1232
ReDim Preserve a(12)
```

Mehrdimensionale Felder

Mehrdimensionale Felder werden einfach dadurch erzeugt, dass Sie bei *Dim* mehrere Indizes angeben. Beim folgenden Feld reicht der erste Index von 0 bis 3, der zweite von 0 bis 4, der dritte von 0 bis 5. Insgesamt hat das Feld somit 4*5*6=120 Elemente.

```
Dim d(3, 4, 5) As Integer
```

ReDim kann auch auf mehrdimensionale Felder angewendet werden, allerdings darf dabei nur die Größe der äußersten Dimension verändert werden. (Nach *Dim a(3,4,5)* dürfen Sie also *ReDim a(3,4,6)* ausführen, nicht aber *ReDim a(4,4,5)*.)

Feldgröße ermitteln

Die Eigenschaft *feld.**Rank*** liefert die Anzahl der Dimensionen. (*d.Rank* liefert 3.) Für jede Dimension kann der zulässige Indexbereich mit *feld.**GetLowerBound**(n)* und *feld.**GetUpperBound**(n)* ermittelt werden, wobei *n* die Dimension ist. (Für die erste Dimension gilt *n=0*!) Die Gesamtzahl aller Elemente kann mit *feld.**Length*** ermittelt werden. Die folgenden Zeilen zeigen die Initialisierung eines dreidimensionalen Felds:

```
For i = 0 To d.GetUpperBound(0)
  For j = 0 To d.GetUpperBound(1)
    For k = 0 To d.GetUpperBound(2)
      d(i, j, k) = i * 100 + j * 10 + k
    Next
  Next
Next
```

Statt *GetUpper/LowerBound* können Sie auch die VB-Funktionen ***UBound*** bzw. ***LBound*** verwenden. Die gewünschte Dimension geben Sie im zweiten Parameter an, wobei die Zählung aber nun mit 1 beginnt! Für die erste Dimension können Sie auf den zweiten Parameter verzichten, es gilt der Standardwert *n=1*.

```
For i = 0 To UBound(d)
  For j = 0 To UBound(d, 2)
    For k = 0 To UBound(d, 3)
      ...
```

* * * TIPP

*Bei Feldern, die Sie selbst mit **Dim** erzeugt haben, können Sie auf die Auswertung von **GetLowerBound** verzichten: Visual-Basic-Felder beginnen immer mit dem Index 0. Bei Feldern, die von anderen Bibliotheken stammen, ist eine Auswertung aber sehr wohl sinnvoll, weil .NET grundsätzlich auch Felder mit einem von 0 abweichenden Startindex unterstützt.*

For-Each-Schleifen

Eine besonders einfache Form, alle Elemente eines Felds zu durchlaufen, bilden die in Abschnitt 5.2 vorgestellten *For-Each*-Schleifen. Die Schleifenvariable muss dabei denselben Datentyp wie das Feld aufweisen. (Durch die Anweisung *Console.WriteLine* wird der Inhalt von *i* in einem Konsolenfenster angezeigt.)

```
Dim c() As Integer = {7, 12, 39}
Dim i As Integer
For Each i In c
  Console.WriteLine(i)
Next
```

Felder löschen

Mit ***Erase*** können Sie alle Elemente eines Felds löschen und den so reservierten Speicher wieder freigeben.

```
Erase a, b, c, d
```

Feldelemente löschen

*Array.**Clear*** setzt eine vorgegebene Anzahl von Elementen auf *0*, *Nothing* oder *False* (je nach Datentyp des Elements). Bei einem *Integer*-Feld entspricht *Array.Clear(f, 7, 2)* den Anweisungen *f(7)=0* und *f(8)=0*.

Felder kopieren

Mit ***Clone*** können Sie eine vollständige Kopie eines Felds erzeugen. *Clone* liefert allerdings ein *Object*-Feld, das mit ***CType*** in ein Feld des gewünschten Typs umgewandelt werden muss. Die folgenden Zeilen zeigen die Anwendung der Methode. Wenn Sie anschließend *b(3)* auswerten, enthält dieses Element erwartungsgemäß den Wert 5. (*CType* wird in Abschnitt 4.12 beschrieben.)

```
Dim a(10) As Integer
Dim b() As Integer
a(3) = 5
b = CType(a.Clone, Integer())
```

Wenn Sie nicht einfach alle Elemente kopieren möchten, nehmen Sie die Methode *Array.**Copy*** zu Hilfe. Das folgende Beispiel kopiert aus dem Feld *c* sechs Elemente nach *d*, wobei das Kopieren in *c* beim Element *2* und in *d* beim Element *0* beginnt. Die *Copy*-Anweisung entspricht also *d(0)=c(2)*, *d(1)=c(3)*, *d(2)=c(4)* etc. Im Konsolenfenster werden daher die Werte 2, 3, 4, 5, 6 und 7 ausgegeben.

```
Dim c(10) As Integer
Dim d(5) As Integer
Dim i As Integer
For i = 0 To 10
  c(i) = i
Next
Array.Copy(c, 2, d, 0, 6)
For i = 0 To 5
  Console.WriteLine(d(i))
Next
```

* * * TIPP

*Wenn Sie ein Feld kopieren, das Objekte von Referenztypen enthält (keine **ValueType**-Daten), dann wird ein sogenanntes **shallow copy** durchgeführt. Das bedeutet, dass nur die Referenzen kopiert werden, dass aber von den Objekten selbst keine Kopie erstellt wird. Beide Felder verweisen dann auf dieselben Objekte.*

Reihenfolge der Elemente vertauschen

*Array.**Reverse*** dreht die Reihenfolge der Elemente um. Das letzte Element eines eindimensionalen Felds wird damit zum ersten (und umgekehrt).

```
For i = 0 To 9
  e(i) = i
Next
Array.Reverse(e)  'nun gilt e(0)=9, e(1)=8 etc.
```

Felder sortieren und durchsuchen

*Array.**Sort*** sortiert das als Parameter übergebene Feld. Durch optionale Parameter kann auch nur ein Teil des Felds sortiert werden. Ein Sortieren ist nur möglich, wenn die im Feld gespeicherten Objekte (z.B. Zahlen, Zeichenketten) vergleichbar sind.

Wenn Sie ein bestimmtes Element in einem Feld suchen, müssen Sie in der Regel alle Elemente durchlaufen. Wenn das Feld aber bereits sortiert ist, können Sie die Suche ganz wesentlich beschleunigen, indem Sie die Methode ***BinarySearch*** zu Hilfe nehmen. Diese Methode benötigt beispielsweise zur Suche in einem Feld mit 1000 Einträgen nur maximal zehn Vergleichsvorgänge. *BinarySearch* liefert als Ergebnis entweder die positive Indexnummer des gefundenen Elements oder eine negative Nummer, wenn das Element nicht gefunden wurde.

```
' Beispiel variablen\felder

Sub sort_array()
  Dim i As Integer
  Dim n As String    'einige VB-Autoren ...
  Dim s() As String = _
    {"Holger Schwichtenberg", "Frank Eller", "Dan Appleman", "Brian Bischof", _
     "Gary Cornell", "Jonathan Morrison", "Andrew Troelsen"}

  ' sortieren
  Array.Sort(s)

  ' suchen
  i = Array.BinarySearch(s, "Dan Appleman")
  Console.WriteLine("Suche nach Dan Appleman: Index = " + i.ToString + " Element = " +
s(i))
End Sub
```

> > > HINWEIS

Generische Methoden zur Bearbeitung von Feldern werden in Abschnitt 7.2 vorgestellt.

Wenn Sie Felder nach eigenen Sortierkriterien ordnen möchten, können Sie an **Sort** *ein Objekt übergeben, dessen Klasse die Schnittstelle* **Collections.IComparer** *realisiert: Dann werden bei jedem Vergleichsvorgang zwei Elemente des Felds an die* **Compare***-Funktion dieser Klasse übergeben. Die Funktion vergleicht die beiden Elemente und liefert als Ergebnis -1, 0 oder 1, je nachdem, ob das erste Objekt kleiner, gleich oder größer als das zweite war. Auf diese Weise können Sie die* **Sort***-Methode mit beliebigen Vergleichskriterien verbinden. Beispielsweise könnten Sie das obige* **String***-Feld dann nach Familiennamen sortieren. Hintergrundinformationen und Beispiele zum individuellen Sortieren von Feldern und Aufzählungen finden Sie in Abschnitt 7.6.*

Syntaxzusammenfassung

Felder deklarieren und verwenden	
Dim f(7) As datentyp	deklariert ein eindimensionales Feld mit acht Elementen, die mit *feld(0)* bis *feld(7)* angesprochen werden.
Dim f(n, m, o, p) As datentyp	deklariert ein vierdimensionales Feld.
ReDim [Preserve] f(n)	verändert die Größe des Felds (optional unter Erhaltung des Inhalts).
Erase f	löscht das Feld.
LBound(f,n) *UBound(f, n)*	ermittelt den minimalen bzw. maximalen Index des Felds für die Dimension *n* (mit *n=0* für die erste Dimension).

Eigenschaften und Methoden der Klasse *System.Array*	
*Array.**BinarySearch**(f, suchobj)* *Array.**BinarySearch**(f, suchobj, icompareobj)*	durchsucht das Feld *f* nach dem Eintrag *suchobj*. Die Methode setzt voraus, dass das Feld sortiert ist, und liefert als Ergebnis die Indexnummer des gefundenen Eintrags. Optional kann eine eigene Vergleichsmethode angegeben werden.
*Array.**Clear**(f, n, m)*	setzt *m* Elemente beginnend mit *f(n)* auf *0*, *Nothing* oder *False* (je nach Datentyp des Elements).
*f2 = CType(f1.**Clone**, datentyp())*	weist *f2* eine Kopie von *f1* zu.
*Array.**Copy**(f1, n1, f2, n2, m)*	kopiert *m* Elemente vom Feld *f1* in das Feld *f2*, wobei *n1* der Startindex in *f1* und *n2* der Startindex in *f2* ist.
*f = Array.**CreateInstance**(type, n [,m [,o]])*	erzeugt ein Feld der Größe *(n,m,o)*, wobei in den einzelnen Elementen Objekte des Typs *type* gespeichert werden können.
*f.**GetLowerBound**(n)* *f.**GetUpperBound**(n)*	ermittelt den minimalen bzw. maximalen Index des Felds für die Dimension *n* (mit *n=0* für die erste Dimension).
*f.**GetValue**(n [,m [,o]])*	liefert das Element *f(n, m, o)*.
*f.**Length***	ermittelt die Gesamtzahl der Elemente des Felds.
*f.**Rank***	gibt die Anzahl der Dimensionen an.
*Array.**Reverse**(f)*	vertauscht die Reihenfolge der Elemente des Felds.
*f.**SetValue**(data, n [,m [,o]])*	speichert in *f(n, m, o)* den Wert *data*.
*Array.**Sort**(f [,icompareobj])*	sortiert *f* (unter Anwendung der Vergleichsfunktion des *ICompare*-Objekts).

4.10 Speicherverwaltung

Dieser Abschnitt geht auf einige Interna der Verwaltung von Variablen ein. Dabei werden Kenntnisse in den Grundtechniken der objektorientierten Programmierung vorausgesetzt, die erst im weiteren Verlauf dieses Buchs vermittelt werden. Der Abschnitt richtet sich daher an Leser, die schon etwas Erfahrung mit Visual Basic haben.

Speichernutzung durch Wert- und Referenztypen

Um elementare Datentypen wie *Integer* oder *Double* bzw. einfache Datenstrukturen möglichst effizient verwalten zu können, unterscheidet .NET zwischen Wert- und Referenztypen (bzw. zwischen *ValueType*-Klassen und gewöhnlichen Klassen, siehe auch Abschnitt 4.3). Diese Unterscheidung hat einen ganz wesentlichen Einfluss darauf, wie die Daten im Speicher verwaltet werden:

- **Gewöhnliche Objekte (für Referenztypen)** werden in einem eigenen Speicherbereich, dem sogenannten *heap*, gespeichert. Das ist deswegen sinnvoll, weil derartige Objekte meistens eine variable Größe haben und weil Objekte typischerweise dynamisch während des Programms erzeugt und wieder gelöscht werden.

  Um die Speicherverwaltung im *heap* kümmert sich die .NET-Runtime-Bibliothek. Nicht mehr benötigte Objekte werden bei einer sogenannten *garbage collection* (wörtlich: Müllabfuhr) automatisch wieder freigegeben. Einige Feinheiten der *garbage collection* werden etwas weiter unten beschrieben. Entscheidend für Sie als Programmierer(in) ist an dieser Stelle nur: Sie brauchen sich im Regelfall nicht selbst um die Speicherverwaltung zu kümmern.

  Bei Prozeduraufrufen werden in Parametern Zeiger (Referenzen) auf die Daten übergeben (nicht die Daten selbst). Ebenso wird bei einer Variablenzuweisung (*obj1 = obj2*) nur ein neuer Zeiger (eine neue Referenz) auf das Objekt eingerichtet. *obj1* und *obj2* zeigen nun also auf dasselbe Objekt, dessen Daten sich im *heap* befinden.

- ***ValueType*-Daten (für Werttypen wie *Integer*)** werden dagegen direkt in Datenstrukturen gespeichert (*allocated inline a structure*). Bei Prozeduraufrufen werden die Daten in den Stack kopiert. Bei einer Variablenzuweisung (*obj1 = obj2*) werden die Daten kopiert.

Boxed value types

Aus Effizienzgründen werden *ValueType*-Objekte also anders behandelt als gewöhnliche Objekte. Es gibt aber Fälle, wo *ValueType*-Objekte auch intern wie gewöhnliche Objekte dargestellt werden müssen – beispielsweise, wenn ein *Integer*-Wert in einer als *Object* deklarierten Variable, einem *Object*-Feld oder einer *Collection* gespeichert wird.

Dazu wird am *heap* Speicher für eine Zwischenschicht (einen sogenannten *wrapper*) reserviert. Die Daten des *ValueType*-Objekts werden dorthin kopiert. Dieser Vorgang wird als *boxing* bezeichnet, die resultierenden Daten als *boxed value types*. Die Rückverwandlung in gewöhnliche *ValueType*-Daten heißt dementsprechend *unboxing*.

Garbage collection

Der Begriff *garbage collection* (wörtlich übersetzt: Müllabfuhr) bezeichnet das Aufräumen des Speichers. Die *garbage collection* wird automatisch ausgeführt, wobei Sie als Programmierer normalerweise keinen Einfluss darauf haben, *wann* das passiert. Vielmehr beobachtet die .NET-Bibliothek die Nutzung von Objekten und den Speicherbedarf und entscheidet selbst, wann der Speicher von nicht mehr benötigten Objekten befreit werden muss.

> > > HINWEIS

Die garbage collection gilt ausschließlich für den heap, auf dem Objekte (Referenztypen) gespeichert werden. **Value-Type***-Daten sind von einer garbage collection nicht betroffen.*

Das .NET-Framework unterscheidet zwischen zwei Typen von Klassen: solchen mit und solchen ohne **IDisposable***-Schnittstelle. Bei Klassen, die* **IDisposable** *implementieren, müssen Sie sich um die Freigabe kümmern, wenn Sie die Objekte nicht mehr benötigen (***Dispose***-Methode)! Die garbage collection kann* **IDisposable***-Objekte nicht bzw. erst später aus dem heap entfernen! Denken Sie also an* **Dispose***, sonst blockiert Ihr Objekt unnötig lange Speicher und womöglich auch andere Ressourcen wie Datenbankverbindungen. Informationen zum richtigen Umgang mit* **IDisposable***-Objekten gibt Abschnitt 6.3.*

Garbage collection manuell auslösen

Normalerweise kümmert sich das .NET-Framework selbstständig darum, eine *garbage collection* durchzuführen, wenn dies notwendig erscheint. In seltenen Fällen kann es sinnvoll sein, die *garbage collection* manuell auszuführen – z.B. wenn Sie wissen, dass Ihr Programm in den nächsten Sekunden keine zeitkritischen Operationen durchführen muss.

Eine *garbage collection* kann unterschiedlich gründlich durchgeführt werden: Durch ***GetTotalMemory*** wird nur eine schnelle, aber eben auch etwas schlampige *garbage collection* ausgelöst. *GC.**Collect**(n)* limitiert die *garbage collection* auf *n* Stufen. Erst *GC.Collect()* führt die *garbage collection* so gründlich wie möglich durch.

Beachten Sie, dass die *garbage collection* in einem eigenen Thread – also quasi parallel zum Hauptprogramm – ausgeführt wird. Die *Collect*-Methode initiiert diesen Vorgang nur; anschließend wird das Hauptprogramm sofort fortgesetzt. Wenn Sie warten möchten, bis die *garbage collection* abgeschlossen ist, müssen Sie anschließend noch *GC.**WaitForPendingFinalizers**()* ausführen.

Speicherverbrauch ermitteln

Mit *GC.**GetTotalMemory**()* können Sie den *heap*-Speicherbedarf ermitteln. An die Methode müssen Sie *True* oder *False* übergeben, je nachdem, ob Sie auf das Ende der durch *GetTotalMemory* ausgelösten *garbage collection* warten möchten oder nicht. Der so ermittelte Wert hat allerdings wenig mit dem tatsächlich vom Programm beanspruchten Speicher zu tun. Insbesondere gibt es Objekte, bei denen zwar einige Verwaltungsinformationen auf dem *heap*, weitere Daten aber in anderen Speicherbereichen abgelegt werden.

Eine Reihe anderer Speicherbedarfparameter können Sie aus den Eigenschaften eines ***Diagnostics.-Process***-Objekts entnehmen. Dazu müssen Sie mit *GetCurrentProcess* ein *Process*-Objekt erzeugen. Anschließend können Sie dessen Eigenschaften auslesen. (Diese Eigenschaften geben eine statische Momentaufnahme wieder. Die Speicherwerte des Objekts verändern sich nur, wenn die Methode *Refresh* ausgeführt wird.)

Mit ***PrivateMemorySize64*** können Sie den Speicherbedarf des Programms ermitteln, sofern der Speicher nicht gemeinsam von mehreren Programmen genutzt wird.

```
Dim pr As Diagnostics.Process
Dim n1, n2 As Integer
pr = Process.GetCurrentProcess()
n1 = pr.PrivateMemorySize64
n2 = pr.PeakVirtualMemorySize64
...
pr.Dispose()
```

Mit dem in Abbildung 4.3 dargestellten Beispielprogramm variablen\garbage-collection können Sie die Mechanismen der Speicherverwaltung experimentell erforschen. Mit den verschiedenen Buttons führen Sie verschiedene speicherintensive Operationen durch. Dabei werden zwar keine Objekte durch *Dispose* freigegeben, es wird aber dreimal pro Sekunde *GetTotalMemory* ausgeführt. Dadurch wird regelmäßig eine oberflächliche *garbage collection* ausgeführt. Mit einem eigenen Button können Sie eine tiefgreifende *garbage collection* explizit auslösen.

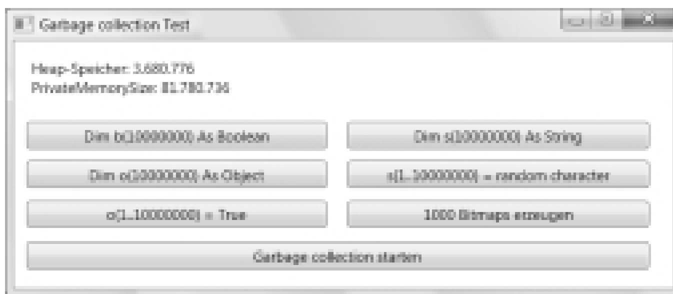

Abbildung 4.3: Speicherbedarf ermitteln

4.11 Datentyp feststellen (GetType)

Dieser Abschnitt stellt einige Methoden vor, mit denen Sie den Typ einer Variable feststellen können. Zu diesem Abschnitt gibt es das Beispielprogramm variablen\var-types, das aber aus Platzgründen und wegen des geringen Informationswerts nicht abgedruckt ist. Sie können das Beispielprogramm als Ausgangspunkt für eigene Experimente mit den hier vorgestellten Methoden verwenden.

In diesem Zusammenhang ist bemerkenswert, dass in Visual Basic jede Variable bzw. jedes Objekt zur Laufzeit (also bei der Programmausführung) weiß, welchen Datentyp sie enthält bzw. von welcher Klasse es abgeleitet ist. Die hier vorgestellten Funktionen helfen nur beim Ermitteln dieser Informationen.

TypeName

Die Visual-Basic-Methode ***TypeName*** liefert eine Zeichenkette mit dem Variablentyp (den Klassennamen) der angegebenen Variable. Für *Dim i As Integer* liefert *TypeName(i)* die Zeichenkette *Integer*.

Bei noch nicht initialisierten *reference*-Objekten liefert *TypeName* als Ergebnis *"Nothing"*. Bei initialisierten *reference*-Objekten liefert *TypeName* den tatsächlichen Datentyp. Wenn Sie also *Dim o As Object* und dann *o=1.5* ausführen, liefert *TypeName* das Ergebnis *"Double"*.

TypeName liefert nur den eigentlichen Klassennamen, nicht aber den Namensraum (also z.B. *"FileInfo"* bei einem Objekt der Klasse *System.IO.FileInfo*).

TypeOf obj Is className

In manchen Fällen ist der Operator ***TypeOf*** eine Alternative zu *TypeName*. *TypeOf* muss immer mit *Is* kombiniert werden. Die Syntax sieht so aus:

```
Dim o As Object
o = ...
If TypeOf o Is System.IO.FileInfo Then
  ...
End If
```

Die obige Abfrage gilt dann als erfüllt, wenn *o* ein Objekt der Klasse *System.IO.FileInfo* oder einer davon abgeleiteten Klasse enthält. Wenn *o* noch nicht initialisiert ist (also *Nothing* enthält), ist die *TypeOf*-Abfrage nicht erfüllt.

Der Vorteil von *TypeOf* besteht darin, dass der Typvergleich viel schneller ausgeführt wird als durch *If TypeName(o)="System.IO.FileInfo"*. Allerdings kann *TypeOf* ausschließlich für Referenztypen eingesetzt werden, nicht aber für Werttypen (*ValueType*-Klassen).

Bei Objekten, deren Klasse aufgrund von Vererbung auf anderen Basisklassen aufbaut, kann *TypeOf* auch zum Test dieser Basisklassen verwendet werden. Das folgende Beispiel basiert auf einem Objekt der Klasse *System.IO.FileInfo*, deren Klassenhierarchie hier dargestellt wird.

Klassenhierarchie für *System.IO.FileInfo*	
Object	.NET-Basisklasse
└─ *MarshalByRefObject*	Objekt nur als Referenz an andere Rechner weitergeben
└─ *IO.FileSystemInfo*	Basisklasse für *DirectoryInfo* und *FileInfo*
└─ *IO.FileInfo*	Informationen über Dateien ermitteln

Aufgrund dieser Hierarchie sind alle vier folgenden *If*-Abfragen erfüllt:

```
Dim o As Object
o = New IO.FileInfo("c:\readme.txt")
If TypeOf o Is Object Then ...
If TypeOf o Is MarshalByRefObject Then ...
If TypeOf o Is IO.FileSystemInfo Then ...
If TypeOf o Is IO.FileInfo Then ...
```

Da jede Klasse von der Basisklasse *Object* abgeleitet ist, liefert *TypeOf o Is Object* immer *True*!

> > > HINWEIS

TypeOf kann nicht mit IsNot kombiniert werden. Statt If TypeOf o IsNot ... müssen Sie die Abfrage in der Form If Not (TypeOf o Is ...) formulieren.

IsArray, IsDate, IsNumeric, IsReference etc.

Visual Basic kennt eine Reihe von *IsXxx*-Methoden, die bei der Klassifizierung von Objekten bzw. Daten helfen.

Methoden zur Objektklassifizierung (Klasse *Microsoft.VisualBasic.Information*)	
IsArray(var)	liefert *True*, wenn die Variable ein initialisiertes Feld enthält. (Das Feld muss mit der Elementzahl deklariert sein. Für *Dim var() As Short* liefert die Methode *False*.)
IsDate(var)	liefert *True*, wenn die Variable als *Date*-Variable deklariert ist oder wenn es sich um eine Zeichenkette handelt, die in der aktuellen Ländereinstellung in ein Datum oder in eine Zeit umgewandelt werden kann.
IsError(var)	liefert *True*, wenn *var* ein Objekt einer Klasse ist, die von *System.Exception* abgeleitet ist. (Derartige Objekte werden in Visual Basic zur Darstellung von Fehlern verwendet – siehe Kapitel 8.)
IsNothing(var)	liefert *True*, wenn *var* ein nicht initialisiertes *reference*-Objekt ist. Bei Variablen für Werttypen liefert *IsNothing* immer *False* (d.h., auch *IsNothing(0)* oder *IsNothing("")* liefert *False*).
IsNumeric(var)	liefert *True*, wenn *var* als *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Decimal*, *Single* oder *Double* deklariert ist oder wenn *var* eine Zeichenkette enthält, die als Zahlenwert interpretiert werden kann (z.B. "*123*").
IsReference(var)	liefert *True*, wenn die Variable ein initialisiertes *reference*-Objekt enthält. Wenn *var* ein Objekt eines Werttyps (*ValueType*-Objekt) oder *Nothing* enthält, liefert die Methode *False*.

GetType

Wenn Ihnen die oben beschriebenen Methoden zu wenige Informationen geben, können Sie mit der Methode ***GetType*** ein Objekt der Klasse *System.**Type*** ermitteln.

```
Dim t As Type
t = variable.GetType()
```

Das Objekt *t* gibt nun Auskunft über unzählige Details der zugrunde liegenden Klasse: *t.name* liefert den eigentlichen Klassennamen, *t.FullName* den vollständigen Namen (Namensraum plus Klassenname), *t.AssemblyQualifiedName* liefert Informationen über die Bibliothek, aus der die Klasse stammt, *t.IsValueType* gibt an, ob es sich um ein gewöhnliches Objekt oder um einen Werttyp handelt, etc.

Statt mit *var.GetType()* können Sie ein *Type*-Objekt für eine bestimmte Klasse auch durch *Type.GetType("klassenname")* ermitteln. *Type.GetType("Integer")* liefert ein *Type*-Objekt, das die *Integer*-Klasse beschreibt.

> > > HINWEIS

*Die **Type**-Klasse kennt Dutzende weitere Eigenschaften und Methoden, mit denen Sie alle nur erdenklichen Informationen über die Klasse ermitteln können. Beispielsweise können Sie damit feststellen, welche Methoden es für die Klasse gibt, welche Ereignisse unterstützt werden etc. Bei der Auswertung dieser Informationen helfen die zahlreichen Klassen des **System.Reflection**-Namensraums.*

Beinahe die einzige Information, die Sie leider nicht ermitteln können, ist der Variablenname. Ein Objekt kann keine Informationen darüber liefern, wie die Variable heißt, die auf das Objekt verweist. (Dafür gibt es viele Gründe. Einer besteht ganz einfach darin, dass mehrere Variablen auf ein- und dasselbe Objekt verweisen können.)

```
' Beispiel variablen\var-types

Dim fi As New IO.FileInfo("c:\datei1.txt")
Dim t As Type = fi.GetType()
Console.WriteLine("t.Name: " + t.Name)
Console.WriteLine("t.FullName: " + t.FullName)
Console.WriteLine("t.AssemblyQualifiedName: " + t.AssemblyQualifiedName)
```

Durch die obigen Anweisungen werden im Konsolenfenster die folgenden Informationen ausgegeben:

```
t.Name: FileInfo
t.FullName: System.IO.FileInfo
t.AssemblyQualifiedName: System.IO.FileInfo, mscorlib,
  Version=1.0.3300.0, Culture=neutral, PublicKeyToken=b77a5c561934e089
```

GetType kann auch in der Form *GetType(datentyp)* verwendet werden, also beispielsweise als *GetType(Integer)* oder *GetType(IO.FileInfo)*. *GetType* liefert damit ein *Type*-Objekt des angegebenen Typs bzw. der angegebenen Klasse zurück. *GetType* gilt hier nicht als Methode, sondern als Operator.

4.12 Datentypkonvertierung und Casting

CType wandelt Daten zwischen verschiedenen Klassen bzw. Datentypen. In Visual Basic ist dafür der Operator *CType* vorgesehen, der syntaktisch aber wie eine Funktion aussieht:

```
obj2 = CType(obj1, klassenname)
```

CType wird nur dann fehlerfrei ausgeführt, wenn eine Typumwandlung tatsächlich möglich ist. Grundsätzlich gibt es dabei zwei Fälle:

- Wenn der ursprüngliche und der neue Typ fundamental voneinander abweichen, muss eine Umrechnung bzw. Umwandlung in den neuen Typ durchgeführt werden. Das ist nur dann möglich, wenn die Ausgangsklasse eine entsprechende Umwandlungsmethode vorsieht. Das ist beispielsweise bei den elementaren Datentypen der Fall. Deswegen können Sie beispielsweise eine Integerzahl in eine Zeichenkette umwandeln.

  ```
  Dim i As Integer = 1000
  Dim s As String
  s = CType(i, String)
  ```

- Sehr oft muss der Typ gar nicht geändert werden – es liegt schon der richtige Typ vor. Allerdings ist die Variable in einem übergeordneten Klassentyp, als Schnittstelle oder einfach als *Object* deklariert, weswegen der Compiler nicht weiß, welche Eigenschaften und Methoden zur Verfügung stehen. (Im laufenden Programm weiß das Objekt sehr wohl, welcher Klasse es angehört, aber das nützt dem Compiler nichts.)

  In solchen Fällen bewirkt *obj2 = CType(obj1, klasse)* nur ein sogenanntes *casting*. Die Daten bleiben unverändert, aber für den Compiler ist nun nur exakte Datentyp bekannt.

Am einfachsten ist dieser zweite Fall anhand eines Beispiels zu verstehen. Die Prozedur *write_-duplicate* hat die Aufgabe, den doppelten Wert des übergebenen Parameters auszugeben. Der Parameter ist als *Object* deklariert. Innerhalb der Prozedur wird nun mit *TypeOf* getestet, ob es sich beim Parameter um einen *Integer*-Wert handelt. Wenn das der Fall ist, wird mit *CType* eine Typumwandlung durchgeführt. Damit weiß der Compiler nun, dass *i* eine *Integer*-Variable ist, und kann *i*2* berechnen. Wenn der Parameter eine Zeichenkette enthält, wird diese analog per *CType* einer *String*-Variablen zugewiesen. Anschließend kann die Zeichenkette durch *s+s* verdoppelt werden.

```
Sub write_duplicate(ByVal obj As Object)
  Dim i As Integer
  Dim s As String
  If TypeOf obj Is Integer Then
    i = CType(obj, Integer)
    Console.WriteLine(i * 2)
  ElseIf TypeOf obj Is String Then
    s = CType(obj, String)
    Console.WriteLine(s + s)
  Else
```

```
    Console.WriteLine("invalid type")
  End If
End Sub
```

Da der Parameter von *write_duplicate* als *Object* deklariert ist, kann jeder beliebige Wert an die Prozedur übergeben werden.

```
write_duplicate(3)
write_duplicate("abc")
write_duplicate(0.4)
```

Als Ausgabe erhalten Sie im Konsolenfenster die folgenden drei Zeilen:

```
6
abcabc
invalid type
```

> > > HINWEIS

Im Stichwortverzeichnis finden Sie unter **CType** *einige Verweise zu Beispielen, die eine reale Anwendung dieser Funktion zeigen. Allerdings sind zum Verständnis der Beispiele oft viele Hintergrundinformationen erforderlich, weswegen hier nur ein unrealistisches (aber dafür einfaches) Beispiel präsentiert wurde.*

CType und die Objekthierarchie

Bei Objekten, deren Klassen von anderen Basisklassen abgeleitet sind, kann *CType* eine Umwandlung in all diese Klassen durchführen. Das Gleiche gilt bei Klassen, die verschiedene Schnittstellen implementieren. Durch *CType* können Sie eine Objektvariable so behandeln, als würde das enthaltene Objekt nur die Merkmale dieser Schnittstelle kennen.

Tatsächlich wird das Objekt in beiden Fällen nicht verändert! *CType* ist vielmehr eine Hilfe für den Compiler, damit dieser weiß, welche Eigenschaften und Methoden zur Verfügung stehen. (Wenn Sie aber eine *casting*-Operation durchzuführen versuchen, die unzulässig ist, tritt ein Fehler auf.)

Zur Verdeutlichung wird nochmals ein Objekt der Klasse *IO.FileInfo* verwendet. Die Vererbungshierarchie dieser Klasse ist auf Seite 117 in einem Diagramm dargestellt. In *obj1* wird ein Objekt der Klasse *IO.FileInfo* gespeichert. Wegen der Klassenhierarchie können die folgenden drei Zuweisungen mit *CType* problemlos durchgeführt werden. Anschließend verweisen alle vier Variablen auf dasselbe Objekt!

```
Dim obj1, obj2 As Object
Dim fsi As IO.FileSystemInfo
Dim fi As IO.FileInfo
obj1 = New IO.FileInfo("c:\readme.txt")
fsi = CType(obj1, IO.FileSystemInfo)
fi = CType(obj1, IO.FileInfo)
obj2 = CType(fi, Object)
```

Obwohl also alle vier Variablen auf dasselbe Objekt verweisen, akzeptiert der Compiler die Verwendung von Eigenschaften und Methoden zur Bearbeitung des Objekts nur bei den Variablen, die die richtige Klasse aufweisen. Die Methode *Exists* ist für die Klasse *IO.FileSystemInfo* deklariert, die Methode *DirectoryName* für *IO.FileInfo*. *Exists* kann daher sowohl auf *fsi* als auch auf *fi* angewendet werden, *DirectoryName* dagegen nur auf *fi*.

```
Console.WriteLine(fsi.Exists())
Console.WriteLine(fi.Exists())
Console.WriteLine(fi.DirectoryName)
```

> > > HINWEIS

Weitere Beispiele zu diesem Abschnitt finden Sie im Beispielprogramm variablen\objektkonvertierung*, das hier aus Platzgründen nicht abgedruckt ist.*

Um **CType** *in all seinen Facetten zu begreifen, müssen Sie wissen, was Klassen und Schnittstellen sind. Diese Begriffe werden in Kapitel 6 ausführlich behandelt.*

Informationen zur automatischen und expliziten Konvertierung zwischen verschiedenen elementaren Datentypen gibt Abschnitt 9.7. Dort lernen Sie die Visual-Basic-Funktionen **CBool**, **CByte**, **CChar**, **CDate** *sowie eine ganze Menge weiterer .NET-Methoden kennen, mit denen Sie Umwandlungen durchführen.*

DirectCast und TryCast

Eine Alternative zu den *Cxxx*-Operatoren ist das VB-Schlüsselwort ***DirectCast***. Der wesentliche Unterschied zu *Cxxx* besteht darin, dass *DirectCast* wirklich nur ein *casting* vornimmt, aber keine Konversion durchführt. *CType(1, String)* liefert "1", *DirectCast(1, String)* liefert dagegen einen Fehler. *DirectCast* ist in der Ausführung minimal schneller als *CType*, funktioniert aber nur für Objekte in voneinander abgeleiteten Klassen.

Wenn Sie vorweg überprüfen möchten, ob ein *casting* möglich ist, führen Sie ***TryCast****(obj, datentyp)* aus. *TryCast* liefert entweder das gewünschte Ergebnis oder *Nothing*, falls der gewünschte Datentyp und der Datentyp von *obj* nicht zueinander kompatibel sind. *TryCast* kann allerdings nur für Referenztypen eingesetzt werden. *obj* darf also nicht eine einfache Zahl oder ein Element einer Struktur (*Structure*) sein.