10 Über Structure und den Unterschied zwischen Referenz- und Wertetypen

327 Der Unterschied zwischen Referenz- und Wertetyp

329 Erstellen von Wertetypen mit Structure am praktischen Beispiel

334 Unterschiedliche Verhaltensweisen von Werte- und Referenztypen

341 Performance-Unterschiede zwischen Werte- und Referenztypen

Jetzt, wo Sie sich mit der Erstellung von Klassen schon recht intensiv auseinander gesetzt haben, wird es Zeit, sich mit einer Variation von Klassen auseinander zu setzen, die ebenfalls ein sehr zentraler Bestandteil des Frameworks ist: die so genannten Wertetypen. Zum genauen Verständnis werden wir einen etwas tieferen Einblick in die Verwaltung von Daten im Framework nehmen.

Lassen Sie uns aber zunächst an der Basis beginnen, mit den Fragen: Was ist ein Referenztyp, was ist ein Wertetyp, und wie unterscheiden sich die beiden voneinander?

Der Unterschied zwischen Referenz- und Wertetyp

Das Framework ist so konzipiert, dass es zweierlei Speicherbereiche gibt, in denen Daten für eine Anwendung gespeichert werden:

- Zum einen gibt es, wie Sie im vergangenen Kapitel schon kennen gelernt haben, den so genannten *Managed Heap* (wörtlich: *verwalteter Haufen*), auf dem die Daten von Referenztypinstanzen (also von Klassen) gespeichert werden.
- Zum anderen gibt es den (Prozessor-)*Stack* (etwa: *Prozessorstapel*), auf dem Wertetypen gespeichert werden.

Der Unterschied zwischen den beiden wird vor allen Dingen in ihrem Verhalten beim Kopieren oder Übergeben an Prozeduren deutlich, teilweise auch in der Geschwindigkeit: Denn während der Prozessor für das Abrufen von Daten auf den Arbeitsspeicher Ihres Computers zugreifen muss, befindet sich der Prozessorstack, wie der Name schon sagt, im Prozessor, bzw. der Zugriff erfolgt prozessorintern – zumindest nicht über den First bzw. Second Level Cache hinaus.

Was bedeutet es nun eigentlich genau, wenn Sie eine Objektvariable verwenden? Bei der Beantwortung dieser Frage möchte ich zunächst die primitiven Datentypen ausklammern, die eine Sonderbehandlung in der CLI aus Performance-Gründen genießen. Ich komme darauf später noch einmal zurück.

Wenn Sie eine Klasse instanzieren, dann referenzieren Sie sie für ihre Lebensdauer über eine Objektvariable. Die Objektvariable dient natürlich nur als Abstraktion für Entwickler und stellt das Repräsentativ für einen kleinen Speicherbereich dar, in den der »Wert« einer Variable hineingeschrieben wird. Und genau hier unterscheiden sich Wertetypen und Referenztypen entscheidend, denn: Während es für Wertetypen jeweils den Speicherbereich auf dem Prozessorstack gibt, werden streng genommen zwei Speicherbereiche für Referenztypen reserviert: Die eigentlichen Daten landen auf dem Managed Heap, und zusätzlich sichert die CLI die korrespondierende Adresse (bei einer 32-Bit-Windows-Version) in weiteren vier Bytes auf dem Stack.

Diese Tatsache wiederum hat mehrere Auswirkungen:

- Ein Referenztyp kann *null* (»nall«, englisch ausgesprochen) sein. Anders als der Wert 0 bedeutet *null*, dass kein Speicherbereich auf dem Managed Heap für das Objekt vorhanden ist. Die Objektvariable eines bestimmten Typs ist in diesem Fall zwar deklariert, es gibt allerdings keine konkreten Instanzdaten, auf die sie zeigt. Ein Wertetyp kann niemals *null* sein (»0« aber schon – jedenfalls bei numerischen Variablen), er ist grundsätzlich definiert, denn: Es ist überhaupt kein Zustand denkbar, bei dem eine Wertevariable ohne Daten auf dem Stack sein könnte (denn wenn es keine Daten auf dem Stack gibt, dann gibt es auch keine korrespondierende Variable dazu). In Visual Basic wird der Wert *null* übrigens durch das Schlüsselwort `Nothing` repräsentiert.
- Wenn Sie einen Wertetyp an einen anderen Wertetyp zuweisen, werden die Daten des entsprechenden Stackbereichs in den Stackbereich des anderen Wertetyps kopiert. Das Resultat: Wertetyp Nr. 2 hält ab sofort die gleichen Werte wie Wertetyp Nr. 1. Auch bei Referenztypen kopieren Sie die korrespondierenden Daten auf dem Stack, allerdings hat das ganz andere Auswirkungen: Referenztyp Nr. 2 zeigt anschließend auf Referenztyp Nr. 1, was bedeutet: Sie greifen mit beiden Objektvariablen auf dieselben Daten zu. Das »Gleiche« und das »Selbe« sind – auch wenn es wie schon im letzten Kapitel bemerkt, laut aktuellem Duden keinen Unterschied zwischen den beiden Wörtern mehr gibt – in diesem Fall in ihrer ursprünglichen Bedeutung zu nehmen. Ändern Sie nämlich Wertetyp Nr. 2, zeigt sich Wertetyp Nr. 1 davon unbeeindruckt. Anders ist das bei den Referenztypen: Ändern Sie eine Eigenschaft von Referenztyp Nr. 2, dann ändert sich auch die Eigenschaft von Referenztyp Nr. 1 – und das muss auch so sein: Denn es gibt keine zwei verschiedenen Eigenschaften; da beide Objektvariablen auf den gleichen Speicherbereich im Managed Heap zeigen, gibt es auch nur ein Objekt, dessen Eigenschaften Sie verändern können. Die Variablen bieten Ihnen nur zwei Möglichkeiten, es anzusprechen.
- Wenn Sie einen Wertetyp an eine Methode, also an eine `Sub` oder `Function` übergeben, dann legt die CLI – solange nichts anderes gesagt wird – eine Kopie des Wertetyps an. Ändern Sie die Variable innerhalb der Prozedur, so hat das keine Auswirkungen auf die Variable, die in dem Programmteil verwendet wurde, von dem aus die Prozedur angesprungen wurde. Bei einem Referenztyp verhält es sich im Gegensatz dazu ähnlich wie bei der Variablenzuweisung: Nur die Adresse des Speicherbereichs im Managed Heap wird der Prozedur übergeben; Änderung der Objekteigenschaften in der Prozedur wirken sich sehr wohl auf das Ursprungsobjekt aus (das ja dasselbe ist).
- Durch die andere Speicherverwaltung von Strukturen im Gegensatz zu Klassen sind Strukturen nicht vererbbar. Sie haben also nicht die Möglichkeit, die Elemente einer Struktur in einer andere Struktur mit `Inherits` »zu übernehmen«.

All diese Dinge klingen in der Theorie vielleicht recht abstrakt. Deswegen möchte ich Ihnen im Folgenden ein komplettes Beispiel präsentieren, anhand dessen diese Zusammenhänge deutlich werden:

Erstellen von Wertetypen mit Structure am praktischen Beispiel

Grundsätzlich gilt: Eine Klasse produziert einen Referenztyp; eine Struktur produziert einen Wertetyp. Prinzipiell erstellen Sie eine Struktur genau wie eine Klasse, und beide haben auch ähnliche Fähigkeiten, wie es durch das folgende Beispiel deutlich werden soll:

Zum Szenario: Gerade beim Programmieren kommt es immer wieder vor, dass Sie Zahlen von einem Zahlensystem ins andere konvertieren müssen. Einige Konvertierungen werden vom Framework unterstützt – so können Sie beispielsweise hexadezimale Werte mit der statischen `Parse`-Funktion etwa so

```
Dim EinInteger As Integer = Integer.Parse("FFFF", Globalization.NumberStyles.HexNumber)
```

in eine Dezimalzahl umwandeln; eine andere Möglichkeit besteht durch die Benutzung der `ToInt`-Funktion bzw. der `ToString`-Funktion der `Convert`-Klasse. Mit

```
Dim EinInteger as Integer = Convert.ToInt32(AnderesZahlensystemString, ZahlenSystemInteger)
```

können Sie ebenfalls einen String, der eine Zahl eines anderen Systems beinhaltet, in einen Integer zurückverwandeln. Allerdings können Sie mit `ZahlenSystemInteger` nur eine Zahl des Dual- (Basis 2), des Oktal- (Basis 8), natürlich des Dezimal- (Basis 10) und des Hexadezimalsystems (Basis 16) umwandeln. Die gleichen Beschränkungen gelten für die Umwandlungen eines Integer-Wertes in ein anderes Zahlensystem, der dann in Form eines Strings abgebildet wird. Diese Aufgabe können Sie mit dem Gegenstück durchführen, etwa durch

```
Dim AnderesZahlensystemString as String = Convert.ToString(IntegerUmzuwandeln, ZahlenSystemInteger)
```

um die Integervariable `IntegerUmzuwandeln` in ein anderes Zahlensystem umzuwandeln, das durch `ZahlenSystemInteger` (nur 2, 8, 10 oder 16 sind auch hier wieder gültige Werte) definiert wird.

Zum Glück – denn einen Wertetyp zu schaffen, der beliebige Konvertierungen (nicht nur vom und ins Hexadezimalsystem) beherrscht, ist ein willkommenes, da brauchbares Beispiel.

BEGLEITDATEIEN: Sie finden das folgende Beispielprojekt im Verzeichnis *\VB 2005 - Entwicklerbuch\D - OOP\Kap10\Structure01.*

So könnte Ihnen beispielsweise sogar das 32-Zahlensystem[1] von Nutzen sein: Sie meinen, Sie hätten das nie benutzt? Falsch: Denn wenn Sie Visual Studio selbst installiert haben, dann mussten Sie zur Installation einen Key eingeben, der natürlich mithilfe eines Algorithmus berechnet wurde. Der Algorithmus basiert wahrscheinlich weniger auf dem Hantieren mit Buchstaben, sondern wird – viel wahrscheinlicher – berechnet. Ein `ULong`-Wert (64 Bit, ohne Vorzeichen, neu ab Visual Basic 2005

[1] Bester Name dafür wäre das »Duotrigesimal-System«, für den es jedoch keine historische Absicherung gibt; ohne Gewähr, dass dieser Link zum Zeitpunkt der Drucklegung noch funktioniert, können Sie sich unter dem IntelliLink *D1001* über die Benennung von Zahlensystemen informieren. Kleines Kuriosum am Rande: Das Hexadezimalsystem ist eigentlich recht inkonsistent benannt, denn es wurde aus einem griechischem und einem lateinischem Wortstamm gebildet (»Hexa« griechisch; »decem« lateinisch). Streng genommen müsste es »sedezimal« oder »sexadezimal« heißen. Da die Kurzform für Hexadezimal kurz »Hex« lautet, können wir über den griechisch-lateinischen Mischmasch aber eher dankbar sein.

übrigens), der dieses Seriennummeralgorithmusergebnis speichert, könnte dann beispielsweise auf Basis des Duotrigesimalsystems (32er-System) in eine Zeichenkette umgewandelt werden. Die Zahl

```
9.223.372.036.854.775.807
```

würde in diesem Zahlensystem wie folgt ausschauen

```
7VVVVVVVVVVVV
```

und die Zahl

```
9.153.672.076.852.735.401
```

um ein anderes Beispiel zu zeigen, lautete wie folgt:

```
7U23085PJGBD9
```

Und hier das erklärte Listing der Struktur `NumberSystems`:

```
Public Structure NumberSystems

    Private myUnderlyingValue As ULong
    Private myNumberSystem As Integer
    Private Shared myDigits As Char()

    Shared Sub New()
        myDigits = New Char() {"0"c, "1"c, "2"c, "3"c, "4"c, "5"c, "6"c, "7"c, "8"c, "9"c, "A"c, _
                               "B"c, "C"c, "D"c, "E"c, "F"c, "G"c, "H"c, "I"c, "J"c, "K"c, "L"c, _
                               "M"c, "N"c, "O"c, "P"c, "Q"c, "R"c, "S"c, "T"c, "U"c, "V"c, "W"c, _
                               "X"c, "Y"c, "Z"c}
    End Sub
```

Die Tabelle für die Umwandlung wird in einem statischen Array gespeichert, das im statischen Konstruktor der Struktur initialisiert wird. Der Konstruktor ist überladen: Ihm wird der Wert übergeben, den die Struktur speichert und im Bedarfsfall als `String` mit entsprechenden Numerale des gewünschten Zahlensystems zurückliefern kann. Sie können dem Konstruktor entweder nur einen `Integer`- oder einen `ULong`-Wert übergeben oder einen `ULong`-Wert und einen weiteren Parameter, der bestimmt, in welchem Zahlensystem Sie arbeiten möchten (bis maximal zum 33er-System).

```
    Sub New(ByVal Value As Integer)
        Me.New(CULng(Value), 16)
    End Sub

    Sub New(ByVal Value As ULong)
        Me.New(Value, 16)
    End Sub

    Sub New(ByVal Value As ULong, ByVal NumberSystem As Integer)

        myUnderlyingValue = Value
        If NumberSystem < 2 OrElse NumberSystem > 33 Then
            Dim Up As Exception = New OverflowException _
                ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
            'Kleiner Scherz für die Englisch sprechenden:
            Throw Up
        End If
```

```
        myNumberSystem = NumberSystem

    End Sub

    Public Property Value() As ULong

        Get
            Return myUnderlyingValue
        End Get
        Set(ByVal Value As ULong)
            myUnderlyingValue = Value
        End Set

    End Property
```

Die Value-Eigenschaft dient lediglich dazu, den für die Konvertierung in das jeweilige Zahlensystem gespeicherten Wert neu zu bestimmen oder abzufragen.

```
    Public Property NumberSystem() As Integer
        Get
            Return myNumberSystem
        End Get
        Set(ByVal Value As Integer)
            If Value < 2 OrElse Value > 33 Then
                Dim Up As Exception = New OverflowException _
                    ("Kennziffer des Zahlensystems außerhalb des gültigen Bereichs!")
                Throw Up
            End If
            myNumberSystem = Value
        End Set
    End Property
```

Die NumberSystem-Eigenschaft verwenden Sie, um das Zahlensystem, in das Sie später die Umwandlungen vornehmen wollen, neu zu bestimmen oder abzufragen.

```
    Public Overrides Function ToString() As String

        Dim locResult As String = ""
        Dim locValue As ULong = myUnderlyingValue

        Do
            Dim digit As Integer = CInt(locValue Mod NumberSystem)
            locResult = CStr(myDigits(digit)) & locResult
            locValue \= CULng(NumberSystem)
        Loop Until locValue = 0

        Return locResult

    End Function
```

Die ToString-Methode ist der erste »Dienstleister« für die eigentliche Aufgabe. Sie verfährt nach folgendem Algorithmus, um die Zeichen (die Numeralia) für das eingestellte Zahlensystem zu ermitteln:

Zunächst kopiert sie den Ursprungswert in eine temporäre Variable, um die `Value`-Eigenschaft nicht zu »zerstören« – diese Variable wird im Folgenden nämlich verändert. Nun führt `ToString` eine Restwertdivision mit der `Mod`-Funktion durch. Diese liefert nicht das Ergebnis der Division, sondern den Restwert. Ein Beispiel im 10er System soll verdeutlichen, was gemeint ist:

Wenn Sie den Wert 129 durch 10 teilen, kommt 12 dabei heraus, und es bleibt ein Rest von 9. Genau diese 9 ist aber in diesem Fall wichtig, denn sie entspricht der ersten gesuchten Ziffer der Ergebniszeichenkette (der äußerst rechts stehenden, um genau zu sein). Anschließend wird der Wert durch die Basiszahl des Zahlensystems geteilt – um bei diesem Beispiel zu bleiben also durch 10 – und als Ergebnis kommt 12 dabei raus. Da das Divisionsergebnis (dieses Mal das Ergebnis, nicht der Restwert) größer ist als 0, wiederholt sich der Vorgang. Wieder wird der Divisionsrestwert ermittelt, und der beträgt dieses Mal 2. Die 2 entspricht der zweiten ermittelten Ziffer. Die Schleife wird nun so lange wiederholt, bis alle Ziffern (bzw. Numeralia) bekannt sind. Die Numeralia selbst werden übrigens aus dem `myDigits`-Array gelesen, dem statischen Array, das durch den statischen Konstruktor der Struktur bei ihrer ersten Verwendung angelegt wird.

HINWEIS: Sie finden in diesem Beispiel einige Konvertierungen von Wertetypen in andere mithilfe der `Cxxx`-Operatoren, zu denen das nächste Kapitel mehr Infos gibt. Nur soviel fürs Erste: Eine Konvertierung vom Typ `Integer` in den `ULong`-Datentyp mithilfe von `CULng` ist in der fett geschriebenen Zeile des oben stehenden Codeausschnittes notwendig, damit die Division über die vollen 64-Bit Breite stattfindet. Ohne diese Konvertierung würde Visual Basic eine (standardmäßige) 32-Bit-Integer-Division initiieren, die mit der 64-Bit-breiten `ULong`-Variable `locValue` nicht funktionieren kann.

```
Public Shared Function Parse(ByVal Value As String, ByVal NumberSystem As Integer) As NumberSystems

    'Hier wird der Value zusammengebaut
    Dim locValue As ULong

    For count As Integer = 0 To Value.Length - 1
        Try
            'Aktuellen Zeichen im String, das verarbeitet wird
            Dim locTmpChar As String = Value.Substring(count, 1)

            'Binäresuche anwenden, um das Zeichen im Array zu finden und damit die Ziffernummer
            Dim locDigitValue As Integer = CInt(Array.BinarySearch(myDigits, CChar(locTmpChar)))

            'Prüfen, ob sich das Zeichen im Gültigkeitsbereich befindet
            If locDigitValue >= NumberSystem OrElse locDigitValue < 0 Then
                Dim Up As Exception = New FormatException _
                    ("Ziffer '" & locTmpChar & "' ist nicht Bestandteil des Zahlensystems!")
                Throw Up
            End If

            'Aus der gefundenen Ziffernummer die Potenz bilden, und zum Gesamtwert addieren
            locValue += CULng(Math.Pow(NumberSystem, Value.Length - count - 1) * locDigitValue)

        Catch ex As Exception
            'Für den Fall, dass zwischendurch 'was schiefgeht
            Dim Up As Exception = New InvalidCastException _
                ("Ziffer des Zahlensystems außerhalb des gültigen Bereichs!")
            Throw Up
```

```
            End Try
            'nächstes Zeichen verarbeiten
        Next

        Return New NumberSystems(locValue, NumberSystem)

    End Function
End Structure
```

Die Parse-Funktion ist eine statische Funktion (genau wie die Parse-Funktionen vieler Datentypen in der CLR), und sie dient dazu, eine Zeichenkette, die dem Wert eines bestimmten Zahlensystems entspricht, in einen NumberSystem-Wert umzuwandeln. Prinzipiell arbeitet Sie nach der Formel

$$\text{NumeraleWert} = \text{Ziffernwert x Zahlensystembasiswert}^{\text{Ziffernposition}-1}$$

Die Werte der Ziffern sind dabei durchnummeriert von 0–33 (Ziffern von 0–9, gefolgt vom großbuchstabigen Alphabet von A–Z). Mit einer Schleife iteriert die Funktion dabei von vorne nach hinten durch die Zeichen des umzuwandelnden Strings. Mit der statischen BinarySearch-Funktion der Array-Klasse ermittelt sie dabei den Ziffernwert. Dieser stellt anschließend den Wert eines Multiplikators des Produkts dar. Der andere Multiplikator ergibt sich durch das Potenzieren des Basiswertes des Zahlensystems mit der Ziffernposition. Auch hier soll ein Beispiel helfen, den Algorithmus besser zu verstehen, dieses Mal jedoch mit einer Konvertierung aus dem Hexadezimalsystem:

Gegeben sei die Zahl »F3E«. Um diese umzurechnen, ermittelt die Funktion den Ziffernwert für »F«, der dem Wert 15 entspricht. Da es das dritte Zeichen im String ist (von hinten gesehen), wird 16 mit 2 (es ist Ziffernposition 1) potenziert (ergibt 256) und mit 15 multipliziert – das Ergebnis lautet: 3840. Nun ist das mittlere Zeichen an der Reihe. Der Ziffernwert beträgt 3, der Exponent 1, denn es ist das 2. Zeichen der Zeichenkette. Zwischenergebnis: 48, addiert zum vorherigen Wert ergibt 3888. Was fehlt ist die letzte Ziffer. Der Exponent[2] ist 0, damit entspricht das Produkt dem Ziffernwert (Multiplikation mit eins verändert nichts), und dieser entspricht 14 für das »E«. Die letzte »14« wird zum Zwischenergebnis addiert, und wir haben das Ergebnis 3902.

Zum Projekt: Hauptprogramm und Struktur sind in dem Programmbeispiel in zwei verschiedenen Dateien untergebracht. Ein Doppelklick auf *NumberSystems.vb* im Projektexplorer öffnet den Quellcode der Struktur; *mdlMain.vb* enthält das Modul für das Hauptprogramm.

```
Module mdlMain

    Sub Main()

        Dim locLong As ULong
        locLong = &HFFFFFFFFFFFFFFFFUL
        Dim locNS As New NumberSystems(locLong)
        Console.WriteLine("{0:#,##0} entspricht:", locLong)

        locNS.NumberSystem = 2 : Console.WriteLine("Binär: " & locNS.ToString)
        locNS.NumberSystem = 8 : Console.WriteLine("Oktal: " & locNS.ToString)
        locNS.NumberSystem = 10 : Console.WriteLine("Dezimal: " & locNS.ToString)
        locNS.NumberSystem = 16 : Console.WriteLine("Hexadezimal: " & locNS.ToString)
```

[2] Sie erinnern sich an die Schulzeit? – Jeder Wert potenziert mit 0 ergibt 1. Die Zahl wird dabei durch sich selbst geteilt.

```
        locNS.NumberSystem = 32 : Console.WriteLine("Duotrigesimal: " & locNS.ToString)

        'Der umgekehrte Weg:
        Console.WriteLine()
        Console.WriteLine("Gegenbeispiel:")
        Console.Write("'7U23085PJGBD9' duotrigesimal entspricht dezimal: ")
        locNS = NumberSystems.Parse("7U23085PJGBD9", 32)
        Console.WriteLine(locNS.Value)

        Console.WriteLine()
        Console.WriteLine("Return drücken zum Beenden")
        Console.ReadLine()
    End Sub
End Module
```

Wenn Sie das Programm starten, erhalten Sie folgende Ausgabe:

```
18.446.744.073.709.551.615 entspricht:
Binär: 1111111111111111111111111111111111111111111111111111111111111111
Oktal: 1777777777777777777777
Dezimal: 18446744073709551615
Hexadezimal: FFFFFFFFFFFFFFFF
Duotrigesimal: FVVVVVVVVVVVV

Gegenbeispiel:
'7U23085PJGBD9' duotrigesimal entspricht dezimal: 9153672076852735401

Return drücken zum Beenden
```

Unterschiedliche Verhaltensweisen von Werte- und Referenztypen

Nun haben Sie Ihre erste Struktur entwickelt, aber inwiefern unterscheidet die sich nun von einer Klasse? Nun, wie in der Einführung schon erklärt, speichern Wertetypen – wie beispielsweise die gerade entwickelte `NumberSystems`-Klasse – im Gegensatz zu Referenztypen ihren Inhalt direkt auf dem Prozessorstack und nicht auf dem Managed Heap. Bei der Zuweisung von einem Wertetyp an einen anderen werden die Inhalte (die Daten) also wirklich kopiert, und beide Wertetypen sind anschließend unabhängig voneinander, wie das folgende Beispiel zeigt:

```
'Neuen Wertetyp deklarieren und definieren.
Dim Wertetyp1 As New NumberSystems(10)

'Zweiter Wertetyp wird deklariert durch den ersten definiert.
Dim Wertetyp2 As NumberSystems = Wertetyp1

'Zweiter Wertetyp bekommt anderen Wert.
Wertetyp2.Value = 20

'Erster Wertetyp behält alten Wert.
Console.WriteLine(Wertetyp1.Value)
```

Die Ausgabe dieses Beispiels lautet »10«.

`Wertetyp1` wird `Wertetyp2` zugewiesen. Da es sich bei beiden Variablen um Wertetypen handelt (instanziert aus der Struktur des vorherigen Beispielprogramms), kopiert die CLR den Inhalt von `Wertetyp1` in `Wertetyp2`. Eine anschließende Änderung von `Wertetyp2` hat keine Auswirkungen auf `Wertetyp1`, da beide ihren unabhängigen Speicherbereich auf dem Stack beanspruchen.

Anders sieht das mit Referenztypen bei Klasseninstanzen aus. Im Beispielprogramm befindet sich ebenfalls eine Klasse, die nur Demonstrationszwecken dient und ebenfalls einen `ULong`-Wert speichern kann. Wenn für das gleiche Beispiel ein Referenztyp verwendet wird, bekommen Sie ein völlig anderes Ergebnis, und zwar eines, das Sie vielleicht nicht unbedingt erwarten:

```
'Neuen Referenztyp deklarieren und definieren
Dim Referenztyp1 As New ReferenzTyp(10)
'Zweiter Referenztyp wird deklariert durch den ersten definiert
Dim Referenztyp2 As ReferenzTyp = Referenztyp1

'Zweiter Referenztyp bekommt anderen Wert
Referenztyp2.Value = 20

'Erster damit ebenfalls!!!
Console.WriteLine(Referenztyp1.Value)
```

Die Ausgabe dieses Beispiels lautet »20«.

Dieses Beispiel verwendet ausschließlich Referenztypen. Sie sehen, dass in diesem Beispiel nur eine einzige Instanz der Klasse erstellt und deswegen auch nur einmal der Speicherbereich für die Instanz auf dem Managed Heap reserviert wird. Die zweite Objektvariable wird zwar deklariert, aber die Zuweisung

```
Dim Referenztyp2 As ReferenzTyp = Referenztyp1
```

erstellt keine neue Instanz, sondern weist der Variablen lediglich einen Zeiger auf die schon vorhandene Instanz zu. Aus diesem Grund ändern Sie oberflächlich betrachtet mit dem ersten Objekt auch das zweite Objekt. Die Wahrheit ist aber, es gibt gar kein zweites Objekt. Mit der zweiten Variablen ändern Sie lediglich das einzig vorhandene Objekt, das jetzt durch die Objektvariablen `Referenztyp1` *und* `Referenztyp2` repräsentiert wird.

Das gleiche Verhalten lässt sich beim Übergeben von Wertetypen bzw. Referenztypen an Prozeduren beobachten. Wir fügen dem Modul zwei `Subs` hinzu, die folgende Form haben:

```
Sub NimmtWertetyp(ByVal Wertetyp As NumberSystems)
    Wertetyp.Value = 99
End Sub

Sub NimmtReferenzTyp(ByVal Referenztyp As ReferenzTyp)
     Referenztyp.Value = 99
End Sub
```

Beide `Subs` machen im Prinzip das gleiche – die eine arbeitet jedoch mit Verweis-, die andere mit Wertetypen. Betrachten Sie jetzt den folgenden Codeauszug, der mit Wertetypen arbeitet:

```
        Dim Wertetyp1 as New NumberSystems(10)
        NimmtWertetyp(Wertetyp1)
        Console.WriteLine(Wertetyp1.Value)
```

```
Referenztyp1 as New ReferenzTyp(10)
NimmtReferenzTyp(Referenztyp1)
Console.WriteLine(Referenztyp1.Value)
```

Die erste Ausgabe lautet hier »10«, die zweite »99«. Wenn Sie einen Wertetyp einer Prozedur übergeben, legt die CLR eine Kopie der eigentlichen Daten der Struktur auf den Stack und übergibt sie der aufgerufenen Prozedur. Die Prozedur arbeitet mit der Datenkopie auf dem Stack, und das Verändern der Variablen hat keine Auswirkungen auf die Variable, mit der die Übergabe initiiert wurde.

Anders wiederum ist das bei Referenztypen. Hier übergibt das aufrufende Programm nur einen Verweis auf den Datenbereich im Managed Heap. Es gibt keine Kopie der Klasseninstanz; die Änderung der Daten erfolgt von beiden Objektvariablen und kann auch durch beide reflektiert werden.

Verhalten der Parameterübergabe mit ByVal und ByRef steuern

Es kann wünschenswert sein, einen Wertetypen durch das Schlüsselwort `ByRef` als Referenz einer Prozedur zu übergeben. In diesem Fall werden beim Aufruf nicht wie sonst die Daten selbst auf den Stack kopiert, sondern ein Zeiger auf die entsprechende Speicherstelle im Stack. Änderungen, die an den Daten innerhalb der Prozedur erfolgen, spiegeln sich damit direkt in der ursprünglichen Variable wider. Eine Änderung der Variablen innerhalb der *aufgerufenen* Prozedur wirkt sich also auch in einer Änderung des Variableninhaltes des *aufrufenden* Programmteils aus. Die Abänderung des Codes macht diesen Sachverhalt deutlich:

```
Dim Wertetyp1 as New NumberSystems(10)
NimmtWertetyp(Wertetyp1)
Console.WriteLine(Wertetyp1.Value)
Dim Wertetyp2 as NumberSystems = Wertetyp1
Wertetyp2.Value = 50
Console.WriteLine(Wertetyp1.Value)
.
.
.
Sub NimmtWertetyp(ByRef Wertetyp As NumberSystems)

    Wertetyp.Value = 99

End Sub
```

Konstruktoren und Standardinstanzierungen von Wertetypen

Da es keine Strukturinstanzen ohne Daten gibt, sorgt die CLR übrigens automatisch dafür, dass bei der Definition einer Struktur immer auch eine entsprechende Dateninstanz erstellt wird – ganz gleich, ob Sie *New* zur Instanzierung verwendet haben oder nicht. Auch hier zeigt ein Beispiel, was gemeint ist:

```
Dim EinWert As NumberSystems
Dim EineReferenz As ReferenzTyp

EinWert.Value = 10
EineReferenz.Value = 10
```

Diese Zeilen werden bis auf die letzte anstandslos verarbeitet. Bei der letzten tritt jedoch eine Ausnahme auf, weil Sie versuchen, die Eigenschaft eines Objektes zu verändern, das auf dem Managed Heap gar nicht existiert (siehe Abbildung 10.1).

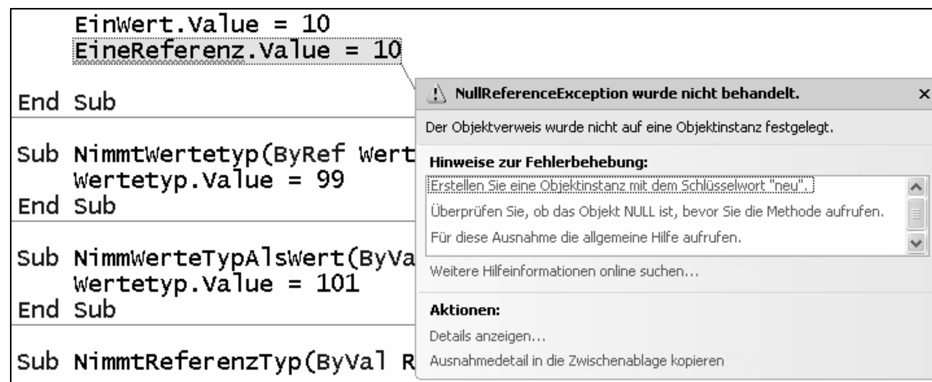

Abbildung 10.1: Versuchen Sie die Eigenschaft eines Objektes zu ändern, das nicht instanziert wurde, erhalten Sie eine Ausnahme

HINWEIS: `ByVal` und `ByRef` haben für die Übergabe von Parametern, bei denen es sich um Referenztypen handelt, übrigens auch Auswirkungen, allerdings nicht beim Verändern des Objektinhalts, sondern beim Neuzuweisen von Objektvariablen an andere Objektinstanzen. Wenn Sie eine Objektvariable mit `ByVal` übergeben, und innerhalb dieser Prozedur der Objektvariablen eine neue Instanz zuweisen, dann ändert sich auch die Instanz, auf die die Objektvariable in der aufrufenden Prozedur zeigt. Mit `ByRef` ist das nicht der Fall.

Keine Standardkonstruktoren bei Wertetypen

Strukturen in VB.NET dürfen also nicht über Standardkonstruktoren verfügen. Ein Standardkonstruktor ist im Falle von VB.NET eine Sub New, die keine Parameter entgegennimmt.

Das folgende Konstrukt ist also beispielsweise fehlerhaft:

```
Public Structure TestValueType

    Private myTestEigenschaft As String

    'Fehler: Ein parameterloser Konstruktor, der nicht als "Shared" deklariert ist,
    'kann nicht in einer Struktur deklariert werden.
    Sub New()
        myTestEigenschaft = "Vorinitialisiert"
    End Sub

    'Das ist OK:
    Sub New(ByVal Vorgabe As String)
        myTestEigenschaft = Vorgabe
    End Sub

    Property TestEigenschaft() As String
        Get
            Return myTestEigenschaft
```

```
        End Get
        Set(ByVal Value As String)
            myTestEigenschaft = Value
        End Set
    End Property

End Structure
```

Strukturen dürfen in VB.NET keine Standardkonstruktoren haben, weil das Framework aus Performancegründen nicht garantieren kann, dass die Konstruktoren auch angewendet werden. Die Instanzierung von Wertetypen, wie sie durch Strukturen definiert werden, geschieht anders als bei »normalen« Klassen, die mit `Class` definiert wurden. Klassen können ausschließlich durch das Schlüsselwort `New` instanziert und anschließend verwendet werden. Strukturen müssen nicht mit `New` vor ihrer ersten Verwendung instanziert werden – vielmehr kann ihre Instanzierung, wie wir schon kennen gelernt haben, implizit erfolgen, also ist beispielsweise das folgende Konstrukt durchaus erlaubt:

```
'Keine Instanzierung mit New...
Dim tvt As TestValueType
'...aber dennoch vorhanden:
tvt.TestEigenschaft = "Test"
```

Sobald ein Wertetyp, wie oben gezeigt, implizit definiert wird, sorgt das Framework dafür, dass alle seine Member-Variablen mit ihren Standardwerten vorbelegt werden (also beispielsweise `0` bei numerischen Typen, `False` für Booleans, etc.). Das geschieht aber nicht notwendigerweise über den – oder vielmehr im – Standardkonstruktor, sondern wird – wie auch immer – von der BCL übernommen. Das bedeutet, dass der Standardkonstruktor bei einer impliziten Verwendung u.U. nicht verwendet wird, und um Fehler von vornherein auszuschließen (nämlich, dass der Entwickler von den Standardwerten abweichende Initialisierungen im Standardkonstruktor vornimmt, die den Member-Variablen aber später gar nicht zugewiesen wurden), verbietet Visual Basic das Vorhandensein von Standardkonstruktoren bei Wertetypen von vornherein.

Gezieltes Zuweisen von Speicherbereichen für Struktur-Member mit den StructLayout- und FieldOffset-Attributen

Strukturen erlauben das gezielte Platzieren von Speicherbereichen für Member-Variablen. Wichtig dafür ist, dass Sie die Struktur mit einem speziellen Attribut markieren und dem Compiler damit »Bescheid geben«, wie er die Speicherbereiche der verschiedenen Member-Variablen überlappen soll.

Mit dieser Möglichkeit wird es zum Kinderspiel, mit den verschiedenen Wertigkeiten der Bytes von Integerwerten zu hantieren. Integer-Zahlen werden, wie Sie sicherlich wissen, aus Bytes »zusammengebaut«. Eine Zahl vom Datentyp `Short` besteht aus 2 Bytes, ein `Integer` aus 4 Bytes und ein `Long` aus 8 Bytes. Bei den vorzeichenbehafteten Integer-Werten wird das höchste Bit für das Vorzeichen verwendet (ist es gesetzt, ist der Wert negativ, ansonsten positiv). Relativ aufwändig ist es, Zahlentypen aus anderen größeren Zahlentypen zu »extrahieren«. Ein `Long` setzt sich beispielsweise aus zwei 32-Bit-Integer-Werten zusammen – dem so genannten höherwertigen DWord (ein Word entspricht einem Byte, ein DWord – Double Word – zwei Words und damit vier Bytes) und dem niederwertigen DWord. Wenn Sie nun wissen möchten, wie der Zahlenwert des höherwertigen DWords lautet, können Sie ihn entweder berechnen oder nur geschickt aus dem Speicher lesen, was natürlich sehr viel schneller geht.

Bleibt die Frage: Wozu brauchen Sie das? 32-Bit-Grafiken beispielsweise speichern pro Pixel drei Farben und einen Alpha-Wert in insgesamt vier Bytes oder als vorzeichenloser 32-Bit-Integer-Wert (UInteger in Visual Basic). Auf die im Folgenden gezeigte Weise könnten Sie beispielsweise eine Struktur schaffen, die auf einem Integer-Wert basiert – und entsprechende Eigenschaften innerhalb der Struktur könnten ohne eine einzige Zeile Rechenaufwand die einzelnen Farbwerte eines Pixels als Byte zurückliefern.

Einige Betriebssystemfunktionen von Windows, die Sie auch durchaus aus Ihren .NET-Programmen heraus aufrufen können (siehe nächster grauer Kasten), verpacken verschiedene Parameter ebenfalls als Kombination in einem einzigen »großbittigen« Datentyp. Der eine Parameter liegt in den höherwertigen Bytes, der andere in den niederwertigen.

Ein Beispiel. Der Long-Wert (vorzeichenlos) $FFFFAABB00FF besteht aus zwei DWords: $0000FFFF sowie $AABB00FF. Um herauszufinden, wie der Wert des oberen DWords lautet, müssten Sie es bei einer Berechnung zunächst mit $FFFFFFFF00000000 ausmaskieren (eine logische AND-Operation durchführen) und dann den kompletten verbleibenden Wert um 32 Bits nach rechts verschieben. Oder aber Sie lesen die oberen 32 Bits direkt aus dem Speicher, weil Sie diesen Teil der Long-Variable auch noch gezielt einer Integer-Variable zuordnen konnten.

Dazu sind zwei Voraussetzungen notwendig: Sie müssen dafür sorgen, dass die Membervariablen einer Struktur ihren benötigten Speicher exakt in der Reihenfolge definieren, in der sie im Quellcode angegeben werden. Dazu dient das StructLayout-Attribut. Und: Sie bestimmen mit dem FieldOffset-Attribut, an welcher Stelle der Speicher einer Member-Variable auf dem Prozessorstack beginnen soll (immer von 0 an gerechnet). Auf diese Weise können Sie dafür sorgen, dass sich Member-Variablen auch verschiedener Typen überlappen, und dann gezielt auf den Speicher der Member-Variablen eines bestimmten Typs, sozusagen unter »vorgegaukeltem Typdeckmantel«, also als anderer Typ zugreifen.

Die prinzipielle Vorgehensweise beim Erstellen einer solchen Struktur zeigt das folgende Beispielprogramm. Mit ihr als Vorlage können Sie sie problemlos für eigene Bedürfnisse erweitern.

BEGLEITDATEIEN: Sie finden das Projekt für dieses Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap10\StructLayout.*

Die Struktur, die dort definiert wird, nennt sich LongEx, und sie erlaubt es, einen 64-Bit-Wert zu definieren, und diesen Wahlweise als vorzeichenbehafteten, vorzeichenlosen oder als Teilwert in Form von höher- und niederwertigen 32-Bit-Integer-Werten (wahlweise vorzeichenbehaften oder vorzeichenlos) abzurufen:

```
'Mitteilen, dass die Reihenfolge der
'Bytedefinitionen strikt einzuhalten ist!
<StructLayout(LayoutKind.Explicit)> _
Public Structure LongEx
    <FieldOffset(0)> Private myUnsignedLong As ULong
    <FieldOffset(0)> Private mySignedLong As Long
    <FieldOffset(0)> Private myLowUnsignedInt As UInteger
    <FieldOffset(4)> Private myHighUnsignedInt As UInteger
    <FieldOffset(0)> Private myLowSignedInt As Integer
    <FieldOffset(4)> Private myHighSignedInt As Integer
```

Mit dem `StructLayout`-Attribut am Anfang der Struktur bestimmen Sie, dass die Reihenfolge der Member-Definition oder besser: die Reihenfolge der Festlegung derer Speicherplätze strikt eingehalten werden soll. Mit dem `FieldOffset`-Attribut erreichen sie anschließend das »Überlappen« der Speicherbereiche für die entsprechenden Member-Variablen. FieldOffset bestimmt durch den angebbaren Parameter, an welcher Speicherstelle im Stack die Daten eines Datentyps abgelegt werden.

WICHTIG: Strukturen können auch Referenztypen aufnehmen. Da in diesem Fall allerdings Zeiger auf die eigentlichen Daten gespeichert werden, wäre die Gefahr groß, dass Sie durch `FieldOffset` einen Zeiger auf einen Referenztyp »verbiegen« – und das verbietet das Framework rigoros, da ein solches Vorgehen nicht nur typunsicheren Code erzeugen sondern das gesamte verwaltete Speicherkonzept über den Haufen werfen würde. Wenn Sie also mit `FieldOffset` arbeiten, dürfen Sie als Member-Variablen auch nur ausschließlich Wertetypen einsetzen – was im Übrigen auch den Einsatz von Arrays ausschließt, da eine Array-Variable auch nur den Zeiger (die Referenz) auf die eigentlichen Array-Daten darstellt.

Die restlichen Eigenschaftenprozeduren dienen dann nur noch dem Auslesen der Member-Variablen auf die gewohnte Weise:

```
Sub New(ByVal Value As ULong)
    myUnsignedLong = Value
End Sub

Sub New(ByVal Value As Long)
    mySignedLong = Value
End Sub

Sub New(ByVal value As UInteger)
    myLowUnsignedInt = value
End Sub

Sub New(ByVal value As Integer)
    myLowSignedInt = value
End Sub

Public Property Value() As ULong
    Get
        Return myUnsignedLong
    End Get
    Set(ByVal value As ULong)
        myUnsignedLong = value
    End Set
End Property

Public ReadOnly Property SignedLong() As Long
    Get
        Return mySignedLong
    End Get
End Property

Public ReadOnly Property HighUnsignedInt() As UInteger
    Get
        Return myHighUnsignedInt
```

```
        End Get
    End Property

    Public ReadOnly Property LowUnsignedInt() As UInteger
        Get
            Return myLowUnsignedInt
        End Get
    End Property

    Public ReadOnly Property HighSignedInt() As Integer
        Get
            Return myHighSignedInt
        End Get
    End Property

    Public ReadOnly Property LowSignedInt() As Integer
        Get
            Return myLowSignedInt
        End Get
    End Property
End Structure
```

Performance-Unterschiede zwischen Werte- und Referenztypen

Ein weiterer wichtiger Unterschied ist die Performance zwischen Werte- und Referenztypen. An die Daten der Wertetypen kommen Sie in der Regel schneller heran, weil ein zusätzlicher Dereferenzierungsschritt nicht erforderlich ist. Erinnern wir uns:

- Wenn Sie mit den Daten eines Wertetyps arbeiten müssen, befinden sich die Daten auf dem Prozessorstack, und der Prozessor kann direkt mit ihnen arbeiten.
- Wenn Sie mit den Daten eines Referenztyps arbeiten wollen, holt sich der Prozessor zunächst die Adressdaten des Speicherbereichs für die eigentlichen Daten im Managed Heap vom Stack. Jetzt erst lädt er die Daten aus dem Managed Heap, indem er die Adresse dereferenziert; dieser Vorgang dauert natürlich entsprechend länger: Zum einen muss der Prozessor einen zusätzlichen Schritt durchführen – nämlich das Dereferenzieren der Adresse. Zum anderen teilt sich das Objekt seinen Datenbereich mit anderen Daten. Wenn Sie sehr große Mengen innerhalb Ihrer Anwendung speichern, ist die Wahrscheinlichkeit natürlich groß, dass der Zugriff auf die Daten physisch im Arbeitsspeicher erfolgt. Liegen die Daten auf dem Stack, ist die Wahrscheinlichkeit größer, dass sie sich im Second- oder sogar First-Levelcache Ihres Prozessors befinden; der Datenzugriff hier erfolgt wesentlich schneller, als auf den Arbeitsspeicher.

HINWEIS: Sie sollten diese Geschwindigkeitsvorteile allerdings nicht als zu groß bewerten, denn: Wenn Sie beispielsweise eine Struktur erstellen, und die Daten, die diese Struktur speichert, in einem Array verwalten, dann verhält sich die Struktur genau wie ein Referenztyp. Array selbst ist nämlich ein Referenztyp, und wenn Referenztypen Daten speichern, die aus Strukturen hervorgehen, verwalten sich diese Strukturinstanzen nahezu wie Referenztypen (der Stack ist nämlich dann nicht mehr erreichbar). Das nächste Kapitel weiß Genaueres zu diesem Thema.

Wieso kann durch Vererbung aus einem Object-Referenztyp ein Wertetyp werden?

Berechtigte Frage. Sie haben seit Beginn des Buches diese Tatsache schon fast als eine Art Dogma kennen gelernt, dass jedes im Framework verwendete Objekt, sei es ein primitiver Datentyp, eine Formular-Komponente, ein Datenbankobjekt, ein Thread etc. von `Object` abgeleitet ist. Dass sich Wertetypen anders verhalten, liegt an der Implementierung der Klasse `ValueType` in der CLR von .NET. Diese wird zwar ebenfalls von `Object` abgeleitet – man könnte aber fast schon sagen »nur pro forma«. Die CLR sorgt nämlich dafür, das ursprüngliche Objektverhalten völlig umzukrempeln. Allerdings geschieht dies nur zum Teil durch Methoden, auf die Sie oder ich ebenfalls zurückgreifen könnten, denn vieles davon geschieht »tief im Inneren« der CLR. Das heißt im Klartext: Wenn Sie eine Struktur in Visual Basic erstellen, dann heißt das für Ihr Objekt, dass es implizit von `ValueType` abgeleitet ist und all seine Fähigkeiten erbt. Neben einer neuen Vorgehensweise, das Objekt anhand seines Inhaltes möglichst eindeutig zu erkennen – die so genannte `GetHashCode`-Funktion wird dabei durch einen neuen Algorithmus ersetzt – ändert sich auch die `Equals`-Methode, die zwei Objekte miteinander vergleicht.

Da `ValueType` eine abstrakte Klasse ist, können Sie sie nicht instanzieren. Sie dient also quasi nur als »Vorlage« für Wertetypen, die Sie aber nicht in eine andere Klasse ableiten, sondern eben mit `Structure` kreieren. Dafür, dass sie innerhalb der CLR einer anderen Speicherverwaltung unterliegt, sorgt dann die CLR intern. Wir »Anwender« haben darauf keinen Einfluss.

HINWEIS: Es gibt viele Fälle, bei denen aus einem Wertetyp ein Referenztyp wird – beispielsweise, wenn Sie Wertetypen in Arrays speichern. Was bei diesem Vorgang des so genannten »Boxing« passiert, dazu gibt das folgende Kapitel nähere Auskunft.