Visual Basic .NET ISBN 3-8272-6041-8

Kapitel 4

VB.NET stellt sich vor

Dieses Kapitel ist sicherlich das für die meisten Visual Basic-Programmierer spannendste Kapitel des Buches (und gleichzeitig auch eines der umfangreichsten). Während es in den ersten beiden Kapitel um die allgemeine Strategie, die Microsoft mit .NET verfolgt und die zwar wichtigen, letztendlich aber doch sekundären Eigenschaften der neuen .NET-Laufzeitumgebung ging, und in Kapitel 3 die zwar ebenfalls wichtige, letztendlich aber nicht ganz so »aufregende« neue Entwicklungsumgebung vorgestellt wurde, geht es in diesem Kapitel (endlich) um das, was viele, viele Visual Basic-Programmierer in den nächsten Wochen und Monaten beschäftigen wird. Die neue Programmiersprache VB.NET. Auch wenn Sie vermutlich schon sehr viel über die spektakulären und teilweise radikal erscheinenden Neuerungen gelesen haben dürften, so viel anders ist das kommende »Visual Basic 7.0« nicht. Vieles ist neu, einiges erscheint aus der Sicht erfahrener Visual Basic-Programmierer tatsächlich ein wenig revolutionär und manches ist ein wenig verwirrend. Nach einer (hoffentlich nicht allzu langen) Eingewöhnungsphase fühlt man sich wieder wohl in dem neuen Zuhause und stellt (vermutlich erleichtert) fest, dass sich soviel gar nicht geändert hat und die neue Art zu Programmieren wirklich gewichtige Vorteile bietet, die sich bereits bei kleinen Programmen bemerkbar machen.

Die Stichworte für dieses Kapitel:

- Ein Wort zur Kompatibilität
- Der neue Projekt- und Programmaufbau
- Die Konsole als universelles Ausgabefenster
- Variablen, Konstanten und Zuweisungen
- Überblick über die Datentypen
- Strings
- Operatoren
- Arrays (Feldvariablen)
- Entscheidungen
- Schleifen
- Benutzerdefinierte Datentypen (Strukturen)
- Parameterübergabe an Funktionen
- Klassen
- Multithreading
- Delegates für Callback-Funktionen und andere Dinge
- Strukturierte Ausnahmebehandlung
- Befehle, die es leider nicht mehr gibt

- Aufruf von API-Funktionen
- Zusammenspiel mit der .NET-Klassenbibliothek
- Zugriff auf COM-Komponenten
- Sonstige Neuerungen
- Die wichtigsten neuen Befehlswörter in der Übersicht

Dieses Kapitel versteht sich als ein allgemeiner Überblick über die Programmiersprache VB.NET. Aus Platzgründen ist es leider nicht möglich, alle Sprachelemente ausführlich vorzustellen. Was bei einem Umstieg von Visual Basic 6.0 auf VB.NET zu beachten ist, wird in Kapitel 10 behandelt. Die Programmierung von Oberflächen mit den neuen Windows Forms-Formularen, die in der Vergangenheit praktisch untrennbar von der Visual Basic-Programmierung war, und die bei VB.NET völlig unabhängig betrachtet werden kann, wird in Kapitel 6 behandelt. Auch dem Thema »Programmierung mit Klassen« wird, weil es so wichtig ist, mit Kapitel 5 ein eigenes Kapitel gewidmet.

4.1 Ein Wort zur Kompatibilität

Liebe Visual Basic-Programmierer! VB.NET ist bei weitem nicht so inkompatibel, wie es vielleicht aufgrund der teilweise recht aufgeregten Diskussionen in den Newsgroups kurz nachdem die erste Beta von VB.NET freigegeben wurde, erscheinen mag. Auch hat Microsoft mit Beta-2 ein paar Kleinigkeiten, wie die Arbeitsweise logischer Operatoren oder den Umstand, dass die Anzahl der Elemente eines neu angelegten Arrays auf 1 und nicht mehr 0 basierte, wieder rückgängig gemacht. Die meisten Umstellungen ergeben sich nicht dadurch, dass einige Befehle eine andere Syntax besitzen oder Operatoren anders arbeiten. Dies sind eher Ausnahmen und keinesfalls die Regel. Die meisten Umstellungen ergeben sich durch den Umstand, dass VB.NET auf .NET und nicht mehr auf COM basiert. Das macht sich im Programmieralltag vor allem an Kleinigkeiten bemerkbar. So werden Visual Basic-Programmierer vermutlich reichlich irritiert feststellen, dass VB.NET scheinbar die Konstante *vbCrLf* nicht mehr kennt. Wie kann eine so elementare Konstante einfach verschwinden? Nun, die Antwort hat etwas damit zu tun, dass diese Konstante, wie viele andere auch, bei Visual Basic 6.0 Teil einer COM-Bibliothek und kein Bestandteil der Sprache selber sind. Natürlich hat Microsoft diese wichtige Konstante nicht einfach gestrichen und den Visual Basic-Programmierern, die sie Hundertfach in ihren Programmen bisher benutzt haben, eine lange Nase gezeigt. Es ist vielmehr so, dass diese Konstanten, und einiges mehr, über die Kompatiblitätsklasse *Microsoft.VisualBasic* zur Verfügung gestellt wird. Wer die beliebte *vbCrLf*-Konstante weiterhin benutzen möchte, muss zu Beginn eines Moduls den Befehl

```
Imports Microsoft.VisualBasic
```

aufführen (das ist eine von mehreren gleichwertigen Möglichkeiten). Dieser Befehl sorgt dafür, dass der Inhalt des angegebenen Namensraums im Programm bekannt ist, und alle seine Mitglieder aufgeführt werden können, ohne dass der komplette Klassenpfad vorangestellt werden muss. Und dann klappt es z.B. auch wieder mit der *vbCrLf*-Konstanten:

```
MsgBox("Yo, das ist eine " & vbCrLf & " zweizeilige Ausgabe!")
```

Ohne den *Imports*-Befehl zu Beginn des Moduls hätte man die Konstante, wie alle übrigen Namen in dem Namensraum auch, wie folgt ansprechen müssen:

```
Microsoft.VisualBasic.vbCrLf
```

Oder man sorgt dafür, dass der Import des Namensraums in den Projekteigenschaften erfolgt. Da dies bei neu angelegten Projekten ohnehin der Fall ist, dürften viele Visual Basic-Programmierer beim Umstieg auf VB.NET den Unterschied gar nicht bemerken. Es ist also alles (wie üblich) nur halb so wild.

> **Tipp**
> Und da wir gerade beim Thema Konstanten sind, die es nur scheinbar nicht mehr gibt. Über die Klasse *Microsoft.VisualBasic.ControlChars* stehen alle wichtigen Ausgabekonstanten (z.B. *Back*, *Cr*, *NullChar* und *Tab*) zur Verfügung.

Noch besser ist es natürlich, sich gar nicht mehr auf die Konstruktionen vergangener Versionen zu verlassen, sondern das Äquivalent der .NET-Klassenbibliothek zu benutzen. Dies wäre im Fall der *vbCrLf*-Konstanten die *NewLine*-Eigenschaft der *Environment*-Klasse:

```
MsgBox("Das ist auch eine " & Environment.NewLine & " zweizeilige Ausgabe!")
```

Diese Variante stellt zudem sicher, dass auch auf anderen Plattformen (welche immer das sein mögen) das gewünschte Resultat entsteht.

> **Hinweis**
> Dass viele VB.NET-Quelltexte keine *Imports*-Befehle enthalten, gewisse Klassen aber dennoch ohne ein Voranstellen der kompletten Namenshierarchie angesprochen werden können, liegt daran, dass sich Imports-Einstellungen auch in den Projekteigenschaften vornehmen lassen. Sie gelten dann für das gesamte Projekt, so dass nicht jedes Dateimodul einen eigenen *Imports*-Befehl enthalten muss.

4.1.1 Der Imports-Befehl

Dieser Befehl dürfte erfahrene Visual Basic-Programmierer zu Recht etwas irritieren. Wird hier wirklich etwas in das Programm importiert? Nein, er stellt lediglich eine Abkürzung für den Gebrauch der Namensräume da (an diesen scheinbar komplizierten Begriff werden Sie sich schnell gewöhnen) Der *Imports*-Befehl importiert den angegebenen Namensraum, so dass alle Unternamen in diesem

Namensraum ohne das Voranstellen des Namensraums aufgeführt werden dürfen. Der *Imports*-Befehl fügt jedoch keine Referenz auf die dahinter stehende Bibliothek ein. Dies muss über den Menübefehl PROJEKT|REFERENZ HINZUFÜGEN unter Umständen nachträglich durchgeführt werden, wenn dies die Visual Studio.IDE nicht bereits erledigt hat.

Code-Beispiel
Das folgende Beispiel geht von einem einfachen Modul aus, das z. B. mit Notepad erstellt und mit dem Kommandozeilencompiler *Vbc.exe* übersetzt werden kann. Außer den Befehlen enthält es keine Referenzen.

Variante a) Ohne den *Imports*-Befehl

```
Module Module1

  Sub Main()
    System.Console.Writeline("Hallo, Welt - wie geht's?")
End Sub

End Module
```

Da *Writeline* im aktuellen Namensraum nicht bekannt ist, muss der komplette Klassenpfad aufgeführt werden.

Variante b) Mit *Imports*-Befehl

```
Imports System.Console

Module Module1

  Sub Main()
    Writeline("Hallo, Welt - wie geht's?")
 End Sub

End Module
```

Durch den *Imports*-Befehl wird der Namensraum *System.Console* importiert und der Compiler veranlasst, die damit verbundene Bibliothek einzubinden. Der Name *Writeline* ist damit im Modul bekannt und muss nicht vollständig qualifiziert werden.

4.2 Der neue Projekt- und Programmaufbau

VB.NET-Programme sind Projekte, die aus einem oder mehreren Dateien bestehen. Bei den Dateien kann es sich um Windows Forms-Formulare, Klassen, Module, Ressourcedateien usw. handeln. Daran hat sich bei VB.NET im Vergleich zu Visual Basic 6.0 nichts geändert. Dennoch gibt es Unterschiede. Die wichtigsten Unterschiede sind:

- Eine Projektdatei definiert nicht automatisch ein ganzes Modul. Vielmehr kann eine Datei mehrere Module und auch mehrere Klassen enthalten, da Klassen und Module über die Befehle *Class/End Class* und *Module/End Module* in allen Projektdateien definiert werden. Es ist daher sogar möglich, ein komplettes Programm, das bei Visual Basic 6.0 aus mehreren Dutzend Projektdateien bestanden hat, in einem einzigen Dateimodul unterzubringen.
- Formulare, Klassen und Module werden standardmäßig mit der Erweiterung *.Vb* gespeichert.
- Verweise (z.B. auf .NET-Klassen oder auf COM-Bibliotheken) werden direkt im Projektmappen-Explorer aufgelistet.
- Ein Projekt kann mehrere Ressourcendateien umfassen.
- Bedingt durch die neuen Projekttypen und erheblich erweiterten Möglichkeiten von Visual Studio .NET gibt es viele neue Modultypen.

Merker
Klassen und Module werden über Befehle und nicht mehr durch Einfügen eines entsprechenden Modultyps definiert.

4.2.1 Der neue Programmaufbau

VB.NET-Module besitzen einen einheitlichen Aufbau. Ein Modul wird durch einen Befehl eingeleitet, der das Modul definiert. Anschließend folgen wie üblich Variablendeklarationen, Konstantendefinitionen, Prozeduren, Funktionen usw. Es gibt praktisch keine Unterschiede zwischen Windows Forms-Formularen, Klassen und Modulen. Das macht die Programmierung sehr flexibel.

Im einfachsten Fall beginnt man ein VB.NET-Programm nicht durch den Start von Visual Studio .NET, der Auswahl eines Projekttyps und gegebenenfalls dem Umschalten auf das Programmcodefenster, sondern indem man in die Eingabeaufforderung umschaltet, Notepad startet, ein paar Zeilen eintippt, die Datei mit der Erweiterung *.Vb* speichert, und den VB.NET-Compiler *Vbc.exe* aufruft. Die resultierende Exe-Datei (es handelt sich um eine so genannte Konsolenanwendung) kann ebenfalls über die Eingabeaufforderung gestartet werden.

Code-Beispiel
Die folgende Befehlssequenz definiert ein Modul, das ohne weitere Änderungen vom VB.NET-Compiler in eine Exe-Datei kompiliert werden kann.

```
Imports System.Console
Module Hauptmodul
Sub Main ()
  Writeline("Die Kreiskonstante Pi ist etwa " & 22 / 7)
End Sub
End Module
```

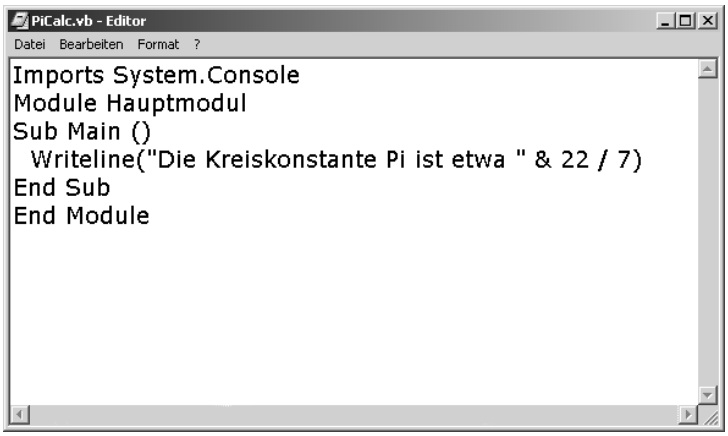

Abbildung 4.1: Ein VB.NET-Programm in der »Entwicklungsumgebung« Notepad

Hinweis
Tritt während der Ausführung einer Konsolenanwendung eine Ausnahme (Laufzeitfehler) auf, erhält man die Möglichkeit den Debugger der Entwicklungsumgebung zu starten.

4.2.2 Der Block als Gültigkeitsbereich

Neben dem Modul und der Prozedur gibt es mit dem *Befehlsblock* einen neuen Gültigkeitsbereich von Variablen. Ein Befehlsblock wird allerdings nicht in geschweifte Klammern eingerahmt. Vielmehr betrachtet VB.NET alle Befehle zwischen *If* und *Else*/*ElseIf* oder *End If* oder *Do* und *Loop* automatisch als einen Befehlsblock. Wird in einem Befehlsblock eine Variable mit dem *Dim*-Befehl deklariert, beschränkt sich ihr Gültigkeitsbereich auf den Befehlsblock. Wird der Befehlsblock verlassen, verliert die Variable aber nicht ihren Wert, sie ist lediglich nicht mehr sichtbar.

Code-Beispiel

```
Imports System.Console
Module Module1
  Private nWert As Long
    Sub Main()
     nWert = 77
     WriteLine(nWert)
     Do
      Dim nWert As Long
      nWert = 99
      WriteLine(nWert)
      If nWert = 99 Then Exit Do
     Loop
    End Sub
End Module
```

Das Beispiel macht deutlich, dass der Wert der Variablen *nWert* bei der Ausgabe in der *Do*-Schleife anders lautet als außerhalb der Schleife, da eine verschiedene Variable angesprochen wird. Übrigens beschränkt sich der Gültigkeitsbereich auf den Schleifenkörper, würde die Variable als Abbruchbedingung des *Loop*-Befehls verwendet, würde sich diese Referenz auf die außerhalb der Prozedur deklarierten Variable beziehen.

Merker
Ein Programmblock definiert einen Gültigkeitsbereich für Variablen.

4.3 Die Konsole als universelles Ausgabefenster

Viele Beispiele in diesem Kapitel sind Konsolenanwendungen, d.h. sie kommen ohne eine Fensteroberfläche aus. Das vereinfacht das Erklären von Programmbefehlen enorm, denn man muss sich keine Gedanken über Ereignisprozeduren oder die Reihefolge, in der einzelne Prozeduren aufgerufen werden müssen, machen. Das allereinfachste VB.NET-Programm besitzt den folgenden Aufbau:

```
Module basSimpel

Sub Main ()
  System.Console.WriteLine ("Das ist aber simpel!")
End Sub
End Module
```

Alle Konsolenwendungen benutzen die *Console*-Klasse für Ausgaben und auch Eingaben. Für Ausgaben sind die Methoden *Write* und *WriteLine*, für Eingaben (hier wartet das Programm darauf, dass in die Eingabeaufforderung Zeichen eingetippt werden) entsprechend die Methoden *Read* und *ReadLine* zuständig.

Tipp
Die *WriteLine*-Methode erlaubt das Einfügen von Platzhaltern in die Ausgabezeichenkette, für die später Werte eingesetzt werden. Die Platzhalter heißen {0}, {1} usw.

Code-Beispiel
```
Dim Feld() As Byte = {12, 33, 44, 55, 77}
Dim n As Integer
For n = 0 To Feld.GetUpperBound(0)
  Console.WriteLine("Der Wert an Position {0} ist {1}", n, Feld(n))
Next n
```

Soll die Eingabeaufforderung nicht gleich wieder geschlossen werden, muss am Ende ein *ReadLine()* folgen.

Hinweis
Was ist aus der guten, alten *InputBox*-Funktion geworden? Nun, sie ist offiziell nicht mehr dabei, steht aber noch über die Kompatibilitätsklasse zur Verfügung.

Abbildung 4.2: Die Eingabeaufforderung ist die Ein- und Ausgabeumgebung für Konsolenanwendungen

4.4 Zuweisungen

Am Prinzip der Zuweisung hat sich bei VB.NET nichts geändert, es gibt aber zwei wichtige Neuerungen:

- Über die neuen Zuweisungsoperatoren kann mit der Zuweisung eine Addition, Subtraktion, Multiplikation oder Division verbunden werden. Das ist eine sehr praktische Abkürzung.
- Es gibt keinen *Set*-Befehl mehr, d.h. Objektvariablen erhalten ihren Wert durch eine normale Zuweisung. Diese mehr als sinnvolle Neuerung dürfte für viele Visual Basic-Programmierer etwas gewöhnungsbedürftig sein.

Merker
Es gibt keinen *Set*-Befehl mehr.

4.5 Variablen und Konstanten

Am Prinzip der Variablen, Konstanten und Operatoren hat sich bei VB.NET nichts geändert, es gibt aber wichtige Neuerungen:

- Die Datentypen basieren (zwangsläufig) auf dem *Common Type System* (CTS). Es gibt daher neue Datentypen (*Chart* und *Short*), Änderungen an vertrauten Datentypen (*Integer* und *Long*), und Datentypen, die nicht mehr vorhanden sind (*Currency* und *Variant*).
- *Decimal* ist ein eigenständiger Datentyp und kein Unterdatentyp von *Variant* mehr.

- Wird bei der Deklaration einer Variablen kein Datentyp angegeben, erhält sie den Datentyp *Object* (dem »Nachfolger« von *Variant*).
- Da es *Variant* nicht mehr gibt, gibt es auch die Spezialwerte *Empty*, *Missing* und *Null* nicht mehr.
- Variablen können mit ihrer Deklaration initialisiert werden. Das ist eine längst überfällige Erweiterung.
- Werden mehrere Variablen mit einem Befehl deklariert, gibt der Datentyp der letzten Variablen den Datentyp der übrigen Variablen vor (das ist eine wichtige Änderung in der »Basic-Tradition« Variablen zu deklarieren)[1].
- Es gibt neue Zuweisungsoperatoren, die gleichzeitig eine Addition oder Subtraktion durchführen.
- Bei den logischen Operatoren wird zwischen logischen Verknüpfungen und bitweisen Verknüpfungen unterschieden.

4.5.1 Namensregeln

An den Namensregeln für Variablen und Konstanten hat sich offenbar nichts geändert. Die Groß-/Kleinschreibung spielt (anders als in C#) nach wie vor keine Rolle.

4.5.2 Neue Datentypen

Mit *Chart* und *Short* gibt es zwei neue Datentypen. Ein Variable vom Typ *Char* nimmt genau ein Zeichen (Unicode-Format) auf. Es handelt sich aber nicht um einen *String*-Datentyp. Die folgende Zuweisung ist daher (bei *Option Strict*) nicht erlaubt, da ein String einem Char zugewiesen werden soll.

Code-Beispiel

```
Dim c As Char
c = "x"
```

Richtig ist:

```
c = CChar("x")
```

Short ist der Nachfolger von *Integer*, so dass eine *Short*-Variable eine Ganzzahl im Bereich -32.768 bis +32.767 aufnehmen kann.

Merker

Short ist der Nachfolger von *Integer*.

1. Und das sorgt dafür, dass Visual Basic-Programmierern allen Pascal-Programmierern wieder etwas selbstbewusster gegenüber treten können.

4.5.3 Geänderte Datentypen

Ein *Integer* umfasst jetzt 4 Byte, ein *Long* entsprechend 8 Byte. Dies sind grundlegende Änderungen, die allerdings auch überfällig waren. Durch den Aktualisierungsassistenten werden die Deklarationen in einem Visual Basic 6.0-Projekt automatisch ausgetauscht, so dass die Änderung in den meisten portierten Projekten nicht spürbar werden sollte.

Als Alternative zu den VB.NET-Datentypen gibt es immer die Möglichkeit, eine Variable direkt mit einem CTS-Datentyp zu deklarieren. Das bietet den Vorteil, etwas eindeutigere Namen verwenden zu können.

Code-Beispiel

```
Dim nKlein As System.int32
```

oder

```
Dim nGross As System.int64
```

Eine Übersicht über alle VB.NET-Datentypen gibt Kapitel 3.

Merker

Ein *Integer* ist 32-Bit (früher 16-Bit) und ein *Long* ist 64-Bit (früher 32-Bit) breit.

4.5.4 Datentypen, die es nicht mehr gibt

Es gibt bei VB.NET keinen *Currency*-Datentyp mehr, der mit 4 festen Nachkommastellen vermutlich auch nicht »eurotauglich« gewesen wäre. Sein Nachfolger ist *Decimal*. Natürlich spricht nichts dagegen, ehemalige *Currency*-Variablen durch *Single*- oder *Double*-Variablen zu ersetzen. Das hängt davon ab, welche Rolle die Genauigkeit spielt. *Decimal*-Variablen bieten zwar die höchste Genauigkeit, könnten ein Programm aber etwas verlangsamen.

Merker

Currency gibt es nicht mehr, sein Nachfolger *Decimal* kann direkt deklariert werden.

4.5.5 Die Deklaration von Variablen

In diesem Punkt war Visual Basic bislang etwas »eigen«. Jeder dürfte wohl die klassische Quiz-Frage kennen, durch die sich sofort feststellen lässt, ob jemand Visual Basic wirklich kennt. Die Frage lautete: »Welchen Datentyp besitzt die Variable *Eins* nach folgender Deklaration?«

```
Dim Eins, Zwei As Long
```

Da kein Datentyp für *Eins* angegeben wurde lautet die Antwort natürlich *Variant* (und nicht *Long* wie in anderen Programmiersprachen). Nun, damit ist bei VB.NET Schluss. Jetzt ist *Long* die korrekte Antwort. Für Variablendeklarationen gelten folgende Besonderheiten:

- Bei der Deklaration von mehreren Variablen wird der Datentyp nur nach der letzten Variablen aufgeführt, die übrigen Variablen erhalten den gleichen Datentyp.

Code-Beispiel

```
Dim nWert1, nWert2, nWert3 As Long
```

- Eine »gemischte« Deklaration von Variablen mit unterschiedlichen Datentypen ist nach wie vor möglich. Man sollte sie aber vermeiden, da dies zu einer deutlich besseren Lesbarkeit des Programmcodes führt.
- Variablen können bereits bei der Deklaration einen Initialisierungswert erhalten, was sehr praktisch ist.

Code-Beispiel

```
Dim nWert1 As Long = 1234, nWert2 As Long= 456, nWert3 As Long
```

In diesem Fall muss jede Variable einen expliziten Datentyp erhalten.

Merker

Variablen können mit ihrer Deklaration initialisiert werden.

4.6 Überblick über die Datentypen

Bereits im letzten Abschnitt wurde angedeutet, dass es bei den Datentypen ein paar Änderungen gegeben hat. Diese Änderungen wurden allerdings nicht mehr oder weniger willkürlich eingefügt. Der Grund für die Änderungen ist, dass die Datentypen von VB.NET als .NET-Programmiersprache mit den durch die *Common Language System* (CLS)-Spezifikation festgelegten Datentypen übereinstimmen müssen. Damit Visual Basic-Programmierern der Umstieg etwas leichter fällt, wurden in den meisten Fällen Aliase (z.B. *Integer* für *Int32* und *Long* für *Int64*) zur Verfügung gestellt, die als Alternative zu den CLS-Datentypen verwendet werden können.

4.6.1 Alles ist ein Objekt

Bei VB.NET ist praktisch alles ein Objekt, auch die einfachen Datentypen, wie z.B. *Integer*, und sogar einfache Zahlen (Literale). Bei allen elementaren Datentypen (mit Ausnahme von *String*) handelt es sich um so genannte *Werttypen*, die auf Strukturen basieren, die sich wiederum von *System.Object* ableiten. Das ist am Anfang ein wenig ungewohnt, macht die Programmiersprache aber sehr viel konsistenter und erleichtert das Einarbeiten.

Code-Beispiel
Der folgende Befehl zeigt, dass sich eine *Integer*-Variable auch wie ein Objekt definieren lässt.

```
Dim i As Integer = 123
Dim j As System.Int32 = 123
Dim k As New Integer()
k = 123
```

Wie sich mit Hilfe des .NET-SDK-Tools *Ildasm.exe* (das Programm disassembliert IL-Code in Quelltext) zeigen lässt, wird in beiden Fällen der gleiche IL-Code generiert (dennoch wird niemand die Variante mit *New* verwenden, da diese – wie bei Strukturen generell – überflüssig ist).

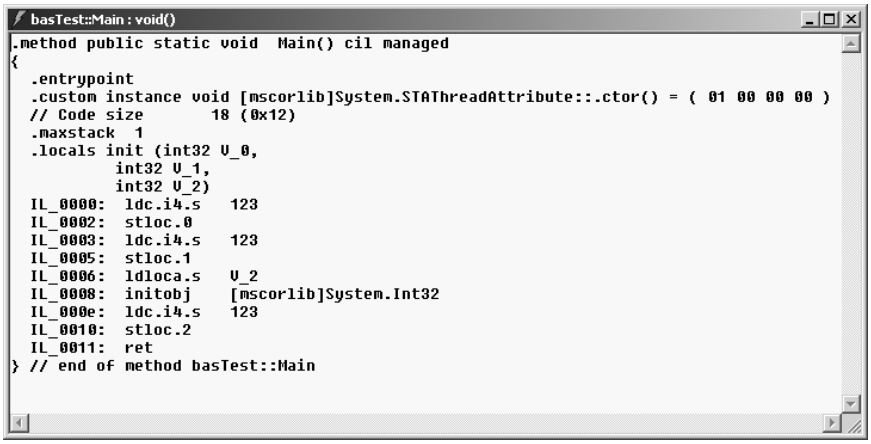

Abbildung 4.3: Der Disassembler zeigt, dass alle drei Deklarationen (fast) identisch sind

Code-Beispiel
In dem folgenden Beispiel wurde die Typenumwandlung absichtlich ein wenig übertrieben. Natürlich ist es nicht notwendig, eine Zahl mit seiner *ToString*-Methode, die die Zahl von *System.Object* erbt, in einen String (genauer, in ein *String*-Objekt) umzuwandeln. Aber es geht und demonstriert, wie ernst es die Microsoft-Entwickler mit der Objektorientiertheit von VB.NET meinen:

```
Dim i, j As System.Int32
i = 4
j = 5
WriteLine("{0}+{1}={2}",i, j, i+j)
Dim s As Integer = i + j
WriteLine(4.ToString + "+" + 5.ToString + "=" + s.ToString)
WriteLine(4 + "+" + 5 + "=" + s.ToString)
```

Tabelle 4.1: Die VB.NET-Datentypen und ihre Pendants in der CLR und in C#

VB.NET-Datentyp	CLR-Datentyp	C#-Pendant	Bedeutung
Byte	Byte	Byte	Ganzzahl zwischen 0 und 255
-	Nicht CTS-kompatibel	sbyte	Ganzzahl zwischen -128 und +127
Short	Int16	short	Ganzzahl zwischen -32.768 und 32.767.
Integer	Int32	int	Ganzzahl zwischen -2.147.483.648 und +2.147.483.647
Long	Int64	long	Ganzzahl zwischen -9.223.372.036.854.775.808 und +9.223.372.036.854.775.807
-	UInt16 (Nicht CTS-kompatibel)	ushort	Vorzeichenlose Ganzzahl zwischen 0 und 65.535
-	UInt32 (Nicht CTS-kompatibel)	uint	Vorzeichenlose Ganzzahl zwischen 0 und 4.294.967.296.
-	UInt64 (Nicht CTS-kompatibel)	ulong	Vorzeichenlose Ganzzahl zwischen 0 und 9.223.372.036.854.775.808.
Single	Single	float	Einfach genaue Fließkommazahl.
Double	Double	double	Doppelt genaue Fließkommazahl.
Boolean	Boolean	boolean	Kann nur die Werte True (-1 bzw. 1) und False (0) annehmen.
Char	Char	char	Unicode 16-Bit-Zeichen.
Decimal	Decimal	decimal	96-Bit-Zahl mit variablem Komma.
String	String	string	Zeichenkette mit einer festen, aber beliebigen Länge.
Object	Object	object	Die »Mutter aller Datentypen«.

4.6.2 CLS und CTS definieren die Datentypen von VB.NET

Da es so wichtig ist, hier noch einmal eine Wiederholung. Die *Common Language Specification* (CLS) legt eine Reihe von Eigenschaften fest, die jede .NET-Programmiersprache erfüllen muss. Damit wird z.B. sichergestellt, dass eine Komponente, die in Programmiersprache X programmiert wurde, in einer Klasse der Programmiersprache Y implementiert werden kann. Ein Teil des CLS ist das *Common Type System* (CTS). Hier sind alle Datentypen zusammengefasst, die jede Programmiersprache unterstützen muss. Zu den CTS-Datentypen gehören *Byte*, *Int16* (*Short*), *Int32* (*Integer*), *Int64* (*Long*), *Single*, *Double*, *Boolean*, *Char*, *Decimal*, *IntPtr* und *String*. VB.NET und C# unterstützen alle CTS-Datentypen, wobei C# zusätzliche Datentypen kennt (die vorzeichenlosen Ganzzahlen), die nicht im CTS (und auch nicht in VB.NET) enthalten sind.

4.6.3 Wert- und Verweistypen

Die Unterscheidung zwischen *Werttypen* (auch Value Types genannt) und *Verweistypen* (Reference Types) ist für alle .NET-Programmierer von großer Bedeutung. Aus Gründen der besseren Performance werden nicht alle Objekte im so genannten *Heap-Arbeitsspeicher* der Anwendung angelegt, der von der Garbage Collection kontrolliert wird. Dies trifft nur für die Verweistypen zu. Ein Verweistyp ist ein Objekt, das eine Referenz (also die Adresse) des Objekts im Arbeitsspeicher beinhaltet. Ein Verweistyp ist ein Objekt, das den Wert des Objekts beinhaltet. Werttypen werden nicht im Heap-Arbeitsspeicher, sondern im Stackspeicher der Anwendung angelegt. Der Zugriff auf diesen Arbeitsspeicher erfolgt etwas schneller. Werttypen sind alle elementaren (primitiven) Datentypen (*Boolean*, *Integer*, *Long*, *Decimal* usw.), sowie Strukturen und Enumerationen. Strings, Arrays und alle übrigen Objekte fallen in die Kategorie der Referenztypen.

4.6.4 Umwandlung von Werttypen in Referenztypen – das Boxing

Werttypen bieten gegenüber Verweistypen unter Umständen einen Geschwindigkeitsvorteil. Werttypen müssen nicht auf dem *Heap* angelegt werden. Objektvariablen, die sich von einem Referenztyp ableiten, speichern eine Referenz auf den Wert, während Objektvariablen, die sich von einem Werttyp ableiten, den Wert des Objekts speichern, was eine Dereferenzierung überflüssig macht. In einigen Fällen kann es notwendig sein (oder sich von alleine ergeben), dass ein Werttyp in einen Referenztyp umgewandelt wird. Dieser Vorgang wird als *Boxing* bezeichnet. Bei dieser Umwandlung wird für den Werttyp ein neues Objekt auf dem Heap angelegt und der Wert des Werttyps hineinkopiert. Anschließend kann das neue Objekt über seine Referenz angesprochen werden. Das Boxing spielt vor allem bei Strukturvariablen eine Rolle, da diese zunächst Werttypen sind, in vielen Fällen aber als Rereferenztyp angesprochen werden.

Code-Beispiel
Das folgende Beispiel legt zuerst einen Werttyp *i* an, der später einem Referenztyp *o* zugewiesen wird (Boxing). Zur Kontrolle gibt die *BaseType*-Eigenschaft des *Type*-Objekts, über das jedes Objekt verfügt, den Klassennamen des Objekts aus (in beiden Fällen *System.ValueType*).

```
imports system.console

Class basTest

Public Shared Sub Main ()
   Dim i As System.Int32
   Dim o As Object
   i = 123
   o = i
   WriteLine("Der Wert von o ist: {0}", o)
   WriteLine("Der Typ von i ist: {0}", i.GetType.BaseType.ToString)
   WriteLine("Der Typ von o ist: {0}", o.GetType.BaseType.ToString)
End Sub

End Class
```

Auch wenn es sich bei der Programmausführung nicht direkt bemerkbar macht, sind Referenztypen bezüglich ihres Laufzeitverhaltens (vermutlich) ein wenig benachteiligt, da der Zugriff auf ihren Wert stets über eine Referenz erfolgt.

Code-Beispiel
Das folgende Beispiel enthält einen weiteren Fall von Boxing (und nimmt bereits einige Befehle vorweg, die erst in den folgenden Abschnitten erklärt werden). Zuerst wird eine Struktur *Point* mit zwei Mitgliedern *x* und *y* angelegt. Strukturen sind Werttypen. Anschließend wird ein *ArrayList*-Objekt mit 10 Feldern angelegt. In einer kleinen Schleife wird in jedes Feld über die *Add*-Methode eine Strukturvariable (Werttyp) eingefügt. Da die *Add*-Methode als Parameter einen Referenztyp erwartet, wird jede Struktur in einen Referenztyp umgewandelt (Boxing) und auf dem Heap angelegt. Das ist grundsätzlich kein Problem, könnte aber in einigen (extremen) Situationen zu Performance-Problemen führen.

```
Imports System.Collections
Imports System.Console

Module basTest
  Structure Point
  Public x As Integer
  Public y As Integer
End Structure

Sub Main()
  Dim PunktFeld As New ArrayList()
  Dim i As Integer
  For i = 0 To 10
```

```
    Dim p As Point
    p.x = i * 10
    p.y = i * 20
' Ein Werttyp wird in einen Referenztyp umgewandelt
    PunktFeld.Add(p)
  Next i
  For i = 0 To PunktFeld.Count - 1
    WriteLine("X:Y-> {0},{1}", PunktFeld(i).x, PunktFeld(i).y)
  Next i
  ReadLine()
End Sub
End Module
```

4.6.5 Strenge Typenüberprüfung

Das Thema »strenge Typenüberprüfung« (engl. »strong typing«) ist zwar untrennbar mit dem Thema »Variablen und deren Deklaration« verbunden, doch da es so wichtig ist, wird ihm ein eigener Abschnitt gewidmet.

Merker
Standardmäßig ist *Option Strict* leider auf *Off* gestellt, d.h. implizite Typenumwandlungen sind zunächst erlaubt. Diese mit Beta-2 eingeführte Änderung ist äußerst problematisch. Sie erleichtert zwar die Konvertierung von Visual Basic 6.0-Programmcode, ist aber ein deutlicher Schritt rückwärts, was die Typensicherheit und damit Robustheit eines Programms angeht[2].

4.6.6 Der neue Option Strict-Befehl

VB.NET unterstützt das so genannte »strong typing«, d.h. es nimmt den Datentyp eines Ausdrucks sehr genau. Visual Basic 6.0 war in dieser Beziehung etwas laxer und erlaubt es z.B., dass ein *String*-Wert einer *Long*-Variablen zugeordnet werden konnte, wenn sich die Zeichenkette in eine Zahl umwandeln ließ. Diese Dinge sind bei VB.NET standardmäßig nicht mehr möglich, wenn die Standardeinstellung *Option Strict On* (die im Quelltext nicht angezeigt wird) nicht ausgeschaltet wird. Der folgende Befehl, der unter Visual Basic 6.0 anstandslos ausgeführt wird, führt unter VB.NET bereits unmittelbar nach der Eingabe zu einem Compilerfehler:

Code-Beispiel

```
Dim sWert As String
sWert = 1234
```

2. Das soll jetzt keine dieser mit erhobenen Zeigerfinger erteilten Verhaltensmaßregeln sein: Aber wir glauben, dass sich alle angehenden VB.NET-Programmierer einen großen Gefallen tun, in dem Sie *Option Strict* auf *On* setzen.

Während diese Zuweisung noch halbwegs plausibel erscheint, ging die Großzügigkeit bei Visual Basic 6.0 noch einen Schritt weiter. Auch die folgende Sequenz wurde ohne Murren akzeptiert:

```
Dim nWert As Long
nWert = "1234"
```

Soweit sollte eine Programmiersprache eigenmächtig nicht gehen, da der Programmierer hier offenbar bewusst eine Zeichenkette gewählt hat und es keinen Sinn ergibt (bzw. ergeben sollte), diese einer *Long*-Variablen zuzuweisen. Auch diese Konstruktion wird von VB.NET nicht akzeptiert.

Damit solche Konvertierungen aber dennoch möglich sind, müssen die zahlreichen (bereits aus Visual Basic 6.0 bekannten) Konvertierungsfunktionen (in anderen Programmiersprachen spricht man von einem »type casting«) zum Einsatz kommen. In diesem Fall erlaubt der Programmierer dem Compiler ausdrücklich eine Typenumwandlung durchzuführen. Die folgende Zuweisung ist auch bei VB.NET in Ordnung:

Code-Beispiel

```
Dim nWert As Long
nWert = CLng("1234")
```

Soll sich VB.NET ähnlich großzügig verhalten wie Visual Basic 6.0, muss die strenge Typenüberprüfung ausgeschaltet werden. Dies kann auf zwei verschiedene Weisen geschehen:

a) in den Projekteigenschaften (in der Rubrik »Erstellen«).

b) zu Beginn eines Moduls über den Befehl

```
Option Strict Off
```

enthalten. Dieser Befehl schaltet die strenge Typenüberprüfung aus.

4.6.7 »Kleinliches« VB.NET

Dass VB.NET was Datentypen angeht, offenbar regelrecht kleinlich werden kann (zumindestens bei *Option Strict On*), macht das folgende kleine Beispiel deutlich.

Code-Beispiel

```
Sub Main
  Dim i, j As Short
  j = 25000
  i = j * 2
  Writeline("Alles ergibt einen Sinn: " & i)
End Sub
```

Frage: Wird der VB.NET-Compiler bei der Kompilierung meckern und wenn ja warum? Antwort: Er wird, denn durch die Multiplikation mit 2 kommt ein *Integer*-Wert ins Spiel und der darf (ohne Typenumwandlung) nicht mit einem *Short*-

Wert multipliziert werden. Abhilfe schüfe die *CShort*-Funktion. Besser wäre es vermutlich, sich darauf zu einigen, die Variablen ebenfalls vom Typ *Integer* zu deklarieren.

4.6.8 Der gute alte Option Explicit-Befehl

Ja, wo ist er denn geblieben? Keine Sorge, auch wenn der beliebte *Option Explicit*-Befehl im Allgemeinen nicht direkt im Quelltext erscheint, er ist nach wie vor vorhanden. In den Projekteigenschaften kann die Einstellung wahlweise auf *On* oder *Off* gestellt werden. In diesem Punkt hat sich bei VB.NET nichts geändert. Wird die Einstellung auf *Off* gesetzt, übersieht es, anders als C#, großzügig nicht deklarierte Variablen.

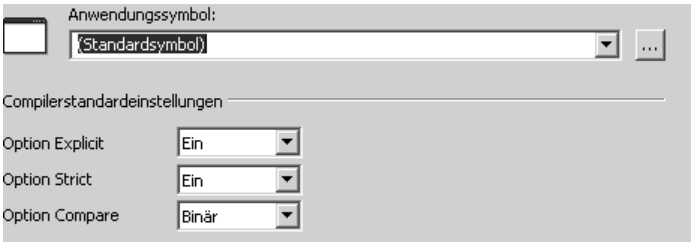

Abbildung 4.4: In den Projekteigenschaften werden die Einstellungen für Option Explicit, Option Strict und Option Compare gesetzt

4.7 Strings

Bei Strings (Zeichenketten) hat sich hinter den Kulissen einiges geändert, wenngleich sich diese Änderungen in der Praxis kaum auswirken, da VB.NET den Programmierern stets mehrere Syntaxalternativen offeriert. Jede Stringvariable ist ein Objekt. Das bedeutet, dass durch die unscheinbare Deklaration

```
Dim sAusgabe As String
```

ein *String*-Objekt definiert wird, wobei es sich bei String um eine .NET-Klasse handelt, die sich von *System.Object* ableitet. Folglich verfügt ein *String*-Objekt über eine Reihe von Eigenschaften und Methoden, die die unter Visual Basic 6.0 benutzten Stringfunktionen (die über die *VisualBasic*-Kompatibilitätsklasse aber nach wie vor zur Verfügung stehen) ablösen.

> **Code-Beispiel**
> Das folgende Beispiel benutzt ein paar der Methoden der *String*-Klasse, um Operationen durchzuführen, die bei Visual Basic 6.0 mit Hilfe der eingebauten Stringfunktionen gelöst werden würden.

```
Dim sName As New String("Ein kleiner Text")
Writeline("Die Länge des Strings: {0}", sName.Length)
Writeline("Die Position des k ist: {0}", sName.Indexof("k"))
WriteLine("Der neue String lautet: {0}", sName.Replace("e", "o"))
WriteLine("Und jetzt alles klein: {0}", sName.ToLower())
```

4.7.1 Strings als unveränderbare Objekte

String-Objekte sind während der Laufzeit nicht änderbar, d.h. wenn einem String ein neuer Wert zugewiesen wird, wird dadurch intern ein neues *String*-Objekt angelegt und diesem der neue Wert zugewiesen:

Code-Beispiel
Das folgende Beispiel veranschaulicht den Umstand, dass durch das Ändern eines Strings ein neues *String*-Objekt angelegt wird. Enthalten die Stringvariablen *sString1* und *sString2* anfänglich noch das gleiche interne *String*-Objekt, ändert sich dies sobald *sString1* einen neuen Wert erhält.

```
Dim sString1, sString2 As String
sString1 = "Hallo, Welt"
sString2 = sString1
WriteLine("String1 = String2: " + (sString1 = sString2).ToString())
WriteLine("String1 Is String2: " + (sString1 Is sString2).ToString())
Mid(sString1, 8, 1) = "Z"
WriteLine("String1 = String2: " + (sString1 = sString2).ToString())
WriteLine("String1 Is String2: " + (sString1 Is sString2).ToString())
Mid(sString2, 8, 1) = "Z"
WriteLine("String1 = String2: " + (sString1 = sString2).ToString())
WriteLine("String1 Is String2: " + (sString1 Is sString2).ToString())
```

Wird das Programm gestartet, erscheint die folgende Ausgabe in der Konsole:

```
String1 = String2: True
String1 Is String2: True
String1 = String2: False
String1 Is String2: False
String1 = String2: True
String1 Is String2: False
```

Auch wenn der =-Operator am Ende wieder feststellt, dass beide Strings den gleichen Inhalt besitzen, macht der *Is*-Operator, der bei VB.NET zwei Objekte vergleicht, deutlich, dass es sich inzwischen um zwei unterschiedliche Objekte handelt.

4.8 Operatoren

Bei den Operatoren sind einige Neuerungen zu vermelden. Es gibt neue Operatoren, die z.B. eine Zuweisung und eine Addition verknüpfen, und bei kombinierten Ausdrücken lässt sich erreichen, dass die folgenden Ausdrücke nicht mehr ausgewertet werden, wenn der erste Ausdruck *False* ergeben hat.

4.8.1 Neue Zuweisungsoperatoren

VB.NET bietet eine Reihe neuer Operatoren, die eine Zuweisung mit einer Rechenoperation verknüpfen. Das ist praktisch, denn dadurch werden Zuweisungen, die mit einem Rechenschritt, etwa dem Erhöhen um 1, verbunden sind, etwas kürzer[3].

Tabelle 4.2: Die neuen Operatoren in der Übersicht

Früher	Neu	Bedeutung
n = n + 1	n+=1	n wird um eins erhöht.
n = n – 1	n-=1	n wird um eins vermindert.
n = n + m	n+=m	n und m werden addiert.
n = n – m	n-=m	m wird von n subtrahiert.
n = n * 4	n*=4	n wird multipliziert.
n = n / 2	n/=2	n wird dividiert.
s1 = s1 & s2	s1&=s2	Zwei Zeichenketten werden verknüpft.

Merker
Eine Variable *x* kann durch die Zuweisung *x+=1* um eins erhöht werden.

4.8.2 Logische Operatoren

Die Bedeutung der logischen Operatoren *AND, OR, XOR* und *NOT* wurde nicht geändert. Dies ist deswegen einen Hinweis wert, weil es in der Beta-1 noch eine Gruppe neuer Operatoren gab (*BitAnd, BitOr, BitXor* und *BitNot*), die speziell für bitweise Verknüpfungen gedacht waren. Diese Operatoren wurden (offensichtlich aufgrund massiver Beschwerden einiger Visual Basic-Enthusiasten) wieder herausgenommen[4].

4.8.3 Auswertung mehrerer logischer Ausdrücke

Besteht ein Gesamtausdruck aus mehreren logischen Ausdrücken, werden wie bei Visual Basic 6.0 die auf den ersten Ausdruck folgenden Ausdrücke auch dann ausgewertet, wenn der erste Ausdruck den Wert *False* zurückgibt (in Beta-1 wurden die folgenden Ausdrücke nicht mehr ausgewertet). Sollte dies nicht erwünscht sein (was in der Regel der Fall sein dürfte), müssen die neuen Operatoren *AndAlso* und *OrElse* zum Einsatz kommen.

3. So weit wie C/C++ und JScript geht VB.NET leider nicht (d.h. es gibt keinen unären ++-Operator, der eine Variable um eins erhöht).
4. Dass eine bekannte Brauerei damit etwas zu tun gehabt haben sollte ist ein unbestätigtes Gerücht.

Code-Beispiel

```
If IsNumeric(oZahl) = True AndAlso oZahl < 0 Then
  Console.WriteLine("Bitte keine negativen Zahlen eingeben")
End If
```

Ist die erste Bedingung nicht erfüllt, wird der folgende Ausdruck nicht mehr ausgewertet (was ansonsten einen Laufzeitfehler zur Folge hätte).

4.9 Feldvariablen (Arrays)

Bei den Feldvariablen (Arrays) gibt es bei VB.NET zwar nur kleinere Änderungen, doch gibt es im Rahmen der *Array*-Klasse der .NET-Klassenbibliothek »Super-Arrays« mit ganz neuen Möglichkeiten. Zunächst zu jenen Feldvariablen, wie sie jeder Visual Basic-Programmierer gewöhnt ist. Hier gibt es bei VB.NET folgende Änderungen:

- Es wird nicht mehr zwischen dynamischen Arrays und Arrays fester Größe unterschieden. Es gibt nur noch dynamische Arrays.
- Arrays können mit ihrer Deklaration initialisiert werden (in diesem Fall darf keine Größe angegeben werden).
- Es gibt keine Möglichkeit mehr, eine untere Feldgrenze festzulegen (es gibt auch keinen *Option Base*-Befehl). Die untere Feldgrenze ist stets 0.
- Hinter den Feldvariablen steckt die *Array*-Klasse der .NET-Klassenbibliothek, die eine Fülle an Möglichkeiten bietet (u. a. eine eingebaute Sortierfunktion).

4.9.1 Initialisierung von Feldvariablen

Auch Feldvariablen können bereits bei der Deklaration einen Wert erhalten. Hier gibt es eine kleine Premiere der besonderen Art. Zum ersten Mal kommen bei Visual Basic nämlich die geschweiften Klammern ins Spiel[5].

Code-Beispiel

```
Private a_iWerte () As Integer = {11, 12, 33}
```

Diese Deklaration funktioniert aber nur, wenn das Array ohne Feldgrenzen deklariert wird.

4.9.2 Die Anzahl der Elemente ermitteln

Als Alternative zur *UBound*-Funktion besitzen Arrays die Eigenschaften *GetLowerBound* (vermutlich nur aus Kompatibilitätsgründen) *GetUpperBound* und *Length*, welches die Anzahl der Elemente in einem Array zurückgibt.

```
WriteLine("Anzahl Elemente: " & a_nWerte.Length)
```

5. Das ist sicherlich nicht unbedingt einer Erwähnung wert, wenngleich wir auf diesen Augenblick ganze 10 Jahren haben warten müssen.

4.9.3 Änderungen am ReDim-Befehl

Der *ReDim*-Befehl kann nicht mehr dazu benutzt werden, eine Variable zu deklarieren. Er beschränkt sich darauf, die Größe einer bereits deklarierten Feldvariablen zu ändern.

4.9.4 Die Array-Klasse

Hinter den Feldvariablen von VB.NET steckt die *Array*-Klasse der .NET-Klassenbibliothek, die eine Vielzahl von Möglichkeiten umfasst und den Umgang mit Arrays auf eine andere, sehr viel professionellere Ebene stellt. Es ist ein Beispiel dafür, wie eng verzahnt die Sprachelemente von VB.NET mit den Klassen der .NET-Klassenbibliothek (zwangsläufig) sind.

Code-Beispiel
Das folgende Beispiel löscht in dem *Array a_nWerte* über die *Clear*-Methode der *Array*-Klasse ab dem 5. Wert genau 2 Elemente.

```
Imports System.Console
Imports Microsoft.VisualBasic.Information
Imports System.Array

Module Modul1

Private a_nWerte () As Long = {11, 12, 33, 44, 55, 66, 77}

Sub Main ()
  WriteLine("Anzahl Elemente: " & a_nWerte.Length)
  WriteLine("Das ist vom Typ: " & Typename(a_nWerte))
  WriteLine("Das 5. Element ist: " & a_nWerte(4))
  System.Array.Clear(a_nWerte, 4, 2)
  WriteLine("Das 5. Element ist: " & a_nWerte(4))
  WriteLine("Das 6. Element ist: " & a_nWerte(5))
  WriteLine("Das 7. Element ist: " & a_nWerte(6))
End Sub

End Module
```

Bei *Clear* handelt es sich, wie bei vielen Methoden der *System.Array*-Klasse um statische Mitglieder, d.h. sie können direkt aufgerufen werden, ohne dass die *Array*-Klasse über eine Objektvariable instanziert werden muss.

4.10 Entscheidungen

Am Prinzip der Entscheidungen hat sich bei VB.NET nichts geändert. Ein *If*-Befehl bleibt ein *If*-Befehl und der Vergleichsausdruck muss auch nicht (wie bei gewissen anderen Programmiersprachen) in Klammern gesetzt werden. Das gilt auch für den *Select Case*-Befehl. Lediglich durch den Umstand, dass durch den Bereich zwischen einem *If*- und einem *Else*-/*ElseIf*- oder *Then*-Befehl ein Gültig-

keitsbereich definiert wird, in dem, wie es in Kapitel 4.2.2 erwähnt wurde, Variablen deklariert werden können, ergibt sich ein kleiner Unterschied zu Visual Basic 6.0.

Merker
Bei *If, Else&Co* hat sich nichts geändert.

4.11 Schleifen

Wie bei den Entscheidungen hat sich auch bei der zweiten Gruppe der »Kernbefehle« praktisch nichts geändert. Dazu sind Schleifenbefehle zu allgemein und zu simpel, als dass sich grundlegend etwas verändern ließe. Es gibt nach wie vor eine *For/Next*-Schleife[6]. Die *While/Wend*-Schleife blieb uns (aus welchen Gründen auch immer, da sie sich immer durch eine *Do*-Schleife ersetzen ließe) erhalten; der abschließende Befehl heißt nun *End While* (einen *End Do*-Befehl gibt es allerdings nicht).

Code-Beispiel
```
While x > 0
' Tue hier irgendetwas
End While
```

Merker
Am Prinzip der Programmschleifen hat sich bei VB.NET nichts geändert. Der *Wend*-Befehl sollte allerdings durch den neuen *End While*-Befehl ersetzt werden.

4.12 Benutzerdefinierte Datentypen (Strukturen)

Ein *benutzerdefinierter Datentyp* ist ein Datentyp, der sich aus verschiedenen Unterdatentypen zusammensetzt, wobei es sich bei einem Unterdatentyp entweder um einen Standarddatentyp oder um einen weiteren benutzerdefinierten Datentyp handelt. Bei VB.NET werden benutzerdefinierte Datentypen über den *Structure*-Befehl eingeleitet und entsprechend über den *End Structure*-Befehl beendet. Außerdem gibt es folgende Unterschiede:

- Die Definition eines benutzerdefinierten Datentyps kann verschachtelt werden, d.h. ein *Structure*-Befehl kann einen weiteren *Structure*-Befehl beinhalten.
- Innerhalb eines *Structure*-Befehls lassen sich auch Funktionen, Eigenschaften und sogar Klassen definieren.

6. Der etwas flexiblere *for*-Befehl aus C#, JScript usw. wurde leider nicht übernommen.

- Zeichenketten fester Länge sind in einer Struktur nicht erlaubt (sie werden allerdings über die Kompatibilitätsklasse doch zur Verfügung gestellt, sie sollten nur nicht mehr bei neuen Projekten auf diese Weise benutzt werden).

Im Grunde entspricht die Definition einer Struktur der einer Klasse, nur dass eine Struktur nicht instanziert werden muss. Außerdem ist eine Struktur ein Werttyp (*Value Type*), während sich es sich einer Klasse stets um einen Referenztyp (*Reference Type*) handelt. Das bedeutet u.a., dass Strukturvariablen im Stackspeicher und nicht in dem von der systemweiten Garbage Collection kontrollierten Heap-Speicher angelegt werden.

Code-Beispiel
Der benutzerdefinierte Datentyp *udtSpieler* sieht in Visual Basic 6.0 wie folgt aus:

```
Type udtSpieler
  Name As String * 32
  SpielerNr As Byte
  HighScore As Byte
End Type
```

In VB.NET lautet die Definition wie folgt:

```
Structure strucSpieler
  Public Name As String
  Public  SpielerNr As Byte
  Public  HighScore As Byte
End Structure
```

Bis auf den Umstand, dass jedem Mitglied ein Gültigkeitsbezeichner (*Private*, *Protected* oder *Public*) vorausgehen muss, und dass es innerhalb einer Struktur keine Zeichenketten fester Länge mehr gibt, hat sich bei dieser simplen Deklaration nicht viel geändert.

Code-Beispiel
Das folgende Beispiel zeigt ein wenig mehr von den Möglichkeiten, die mit Strukturen bei VB.NET einhergehen.

```
Imports System.Console

Structure structSpieler
  Public Name As String
  Public Level As Byte
  Public AnzahlSpiele As Byte
  Private a_bScores() As Byte
  Public Property Score(ByVal SpielNr As Byte) As Byte
   Set
    a_bScores(SpielNr) = Value
   End Set
   Get
     Return a_bScores(SpielNr)
   End Get
```

```
  End Property

  Public ReadOnly Property AnzahlScores() As Integer
    Get
      Return a_bScores.GetUpperBound(0)
    End Get
  End Property

  Sub New(ByVal bAnzahlScores As Byte)
   ReDim a_bScores(bAnzahlScores)
  End Sub
End Structure

Sub Main()
  Dim uSpieler As New structSpieler(10)
  Dim nInx As Long
  With uSpieler
    .AnzahlSpiele = 1
    .Name = "Gerd Müller"
    .Score(0) = 12
    .Score(1) = 18
    .Score(2) = 37
  End With
  For nInx = 0 To uSpieler.AnzahlScores
    If uSpieler.Score(CByte(nInx)) = 0 Then Exit For
    WriteLine("Punktzahl von Spieler " _
     & uSpieler.Name & " in Spiel Nr. " & nInx & ": " _
     & uSpieler.Score(CByte(nInx)))
  Next
  ReadLine()
End Sub
```

Diese Struktur entspricht, vom ihrem Aufbau her gesehen, bereits einer Klasse. Es gibt Eigenschaften, private Elemente und auch einen Konstruktor, der das private Feld initialisiert.

4.13 Eingebaute Funktionen

Ein Pluspunkt von Basic und natürlich auch Visual Basic war der, im Vergleich zu anderen Programmiersprachen, reichhaltige Funktionsumfang. Diesen gibt es auch bei VB.NET. Allerdings stammt ein großer Teil der Funktionen aus der .NET-Klassenbibliothek, was in einigen Fällen dazu führt, das vertraute Funktionen auf einmal andere Namen besitzen (so heißt die *Sqr*-Funktion nun *Sqrt* und ist eine Methode der *Math*-Klasse der .NET-Klassenbibliothek). Zu Inkompatibilitäten bei der Übernahme von Visual Basic 6.0-Projekten führt dies allerdings nur selten, da die Kompatibilitätsklasse fast alle alten Funktionsnamen zur Verfügung stellt (eine der wenigen Ausnahmen ist die *Sqr*-Funktion, die durch die *Sqrt*-Funktion ersetzt werden muss).

Code-Beispiel
Das folgende Beispiel zeigt, wie die neue *Sqrt*-Funktion der .NET-Bibliothek aufgerufen wird.

```
Imports System.Console
Imports System.Math

Module Modul1

Sub Main ()
  Dim snZahl, snErgebnis As Double
  snZahl = 222
  snErgebnis = sqrt(snZahl)
  WriteLine("Ja, die Wurzel, die Wurzel ist: " & snErgebnis)
End Sub

End Module
```

Merker
Viele »eingebaute« VBA-Funktionen wurden durch die Methoden von Klassen in der .NET-Klassenbibliothek ersetzt.

4.14 Funktionen

Funktionen und Prozeduren spielen bei VB.NET die gleiche Rolle wie bei Visual Basic 6.0 und sie werden auch auf die gleiche Weise definiert. Es gibt allerdings wichtige Änderungen was die Parameterübergabe angeht.

4.14.1 Übergabe als Wert oder als Referenz?

In diesem Punkt gibt es bei VB.NET eine ganz wichtige Änderung: Der Default für die Parameterübergabe bei Funktionen und Prozeduren ist nicht mehr *ByRef* (Übergabe als Referenz, also Adresse), sondern *ByVal* (Übergabe als Wert).

Für die Umstellung von Visual Basic 6.0-Projekten bedeutet dies, dass allen Prozedurparametern, denen kein *ByVal* vorausgeht, ein *ByRef* vorangestellt werden sollte.

Referenzübergabe bei Objektvariablen
Die Frage, ob ein Parameter als Wert (*ByValue*) oder als Referenz (*ByRef*) übergeben wird, spielt übrigens auch bei Objektvariable eine Rolle. In beiden Fällen wird zwar eine Referenz auf die Instanz übergeben (*ByVal* bedeutet also nicht, dass die komplette Instanz übergeben wird), bei der Übergabe mit *ByRef* besteht die Möglichkeit, durch Zuweisen einer neuen Referenz an die Prozedurvariable, dem Objekt aus dem aufrufenden Programmteil eine andere Instanz zuzuweisen.

Code-Beispiel
Die folgende Visual Basic 6.0-Befehlsfolge soll den Unterschied zwischen Wert- und Referenzübergabe bei Objektvariablen veranschaulichen. Die Klasse *CTest* enthält dabei eine Eigenschaft *Wert*. Je nach dem, ob die Objektreferenz in *oRef* als Wert oder als Referenz übergeben wird, lässt sich ihr Inhalt in der Prozedur ändern.

```
Private Sub cmdTest_Click()
  Dim oRef As Ctest
  Set oRef = New Ctest
  oRef.Wert = 123
  P1 oRef
  MsgBox oRef.Wert
End Sub

Sub P1(ByRef objRef As CTest)
  Dim oRef2 As Ctest
  Set oRef2 = New Ctest
  oRef2.Wert = 456
  Set objRef = oRef2
End Sub
```

4.14.2 Rückgabewerte von Funktionen

Um den Rückgabewert einer Funktion festzulegen, muss dieser nicht länger dem Funktionsnamen zugewiesen werden. Dafür gibt es bei VB.NET als Alternative den *Return*-Befehl. Auch wenn der Befehl optional ist, sollte er verwendet werden.

Code-Beispiel
```
Function KreisFlaeche (r As Single) As Single
  Dim Pi = 22 / 7 As Single
  Return r ^2 * Pi
End Function
```

4.15 Klassen

Das Thema »Klassen« ist natürlich ein gewichtiges bei VB.NET. Zunächst zwei Dinge vorweg: 1. Klassen gibt es bereits seit Visual Basic 4.0. 2. Am grundsätzlichen Umgang mit Klassen hat sich bei VB.NET nichts geändert. Wer bereits mit der Klassenprogrammierung firm war, wird sich relativ schnell in die Neuerungen einarbeiten. Wer sich noch nie mit der Klassenprogrammierung beschäftigt hat (ähm), wird sich mit dem neuen, sehr viel konsistenteren und sehr viel stärker an die Methodik objektorientierter Programmiersprachen angepasstem Modell schneller zurecht finden (Sie sehen, wir sehen das grundsätzlich optimistisch).

In diesem Abschnitt geht es um einen ersten Überblick. Ein wenig ausführlicher und unterstützt durch ein kleines Beispiel, wird das Thema in Kapitel 5 behandelt.

Tipp
Die am Anfang nicht ganz einfache Klassenprogrammierung lässt sich am einfachsten in einer Konsolenanwendung üben. Die Entwicklungsumgebung bietet allerdings den unschätzbaren Vorteil, dass sie nicht zusammen passende Schlüsselwörter unterstreicht und Fehlermeldungen anzeigt, aus denen sich häufig direkt die Fehlerursache ableiten lässt.

4.15.1 Die Definition einer Klasse

Eine Klasse kann »überall« im Programm über das Befehlspaar *Class*/*End Class* definiert werden. Die denkbar einfachste Klassendefinition sieht daher wie folgt aus.

Code-Beispiel
```
Class CSpieler
' Hier passiert irgendwas
End Class
```

4.15.2 Instanzieren einer Klasse

Klassen werden im Allgemeinen instanziert, damit die resultierenden Objekte in einem Programm eingesetzt werden können (mit den Shared-Mitgliedern stellt VB.NET eine Möglichkeit zur Verfügung Mitglieder aufrufen zu können, ohne dass die Klasse instanziert werden muss). Dies geschieht wie bei Visual Basic 6.0 über das Schlüsselwort *New*, doch hat dieses eine andere und deutlich erweiterte Bedeutung erhalten.

Code-Beispiel
Der folgende Befehl instanziert die *Klasse CSpieler* und weist die resultierende Referenz der Variablen *oTorwart* zu.

```
Dim oTorwart As CSpieler = New CSpieler()
```

Anders als in Visual Basic 6.0 lassen sich auch Objektvariablen in einem Schritt deklarieren und instazieren. Ebenfalls anders als bei Visual Basic 6.0 hat die Verwendung von *New* bei der Deklaration keine unerwünschten Nebeneffekte.

Die Alternative zu dem obigen Befehl besteht darin, die Variable zu erst zu deklarieren und im nächsten Schritt zu instazieren:

```
Dim oTorwart As CSpieler
oTorwart = New CSpieler()
```

Den *Set*-Befehl, der bei Visual Basic 6.0 zwingend erforderlich war, gibt es bei VB.NET nicht mehr.

4.15.3 Implementieren von Eigenschaften und Methoden

Eine Eigenschaft ist eine Variable, nur dass diese zu einer Klasse gehört und damit nur im Zusammenhang mit der Klasse angesprochen werden kann. Methoden sind Funktionen (oder Prozeduren), die in der Klasse definiert sind, und daher ebenfalls nur im Zusammenhang mit der Klasse aufgerufen werden. Eigenschaften und Methoden werden auch als Mitglieder (engl. »members«) der Klasse bezeichnet. Im Vergleich zu Visual Basic 6.0 hat sich bei den Methoden wenig (außer, dass sie überladen und überschrieben werden können – mehr dazu später), bei den Eigenschaften viel geändert. Sie werden nicht nur logischer implementiert, es gibt auch zusätzliche Möglichkeiten spezielle Attribute einer Eigenschaft (etwa eine Nur-Lese-Eigenschaft) zu implementieren.

Code-Beispiel
Das folgende Beispiel implementiert eine Eigenschaft *Name* in der Klasse *CSpieler*[7].

```
Class CSpieler
  Dim m_sName As String
  Property Name() As String
   Get
    Return m_sName
   End Get
   Set
    m_sName = Value
   End Set
   End Property
End Class
```

Der *Property*-Befehl leitet die neue Eigenschaft ein. Anschließend folgen ein *Get*- und ein *Set*-Teil, die beide optional sind. Der *Get*-Teil wird aufgerufen wenn der Wert der Eigenschaft abgefragt wird, der *Set*-Teil wird entsprechend aufgerufen wenn die Eigenschaft ihren Wert erhält. Unter Visual Basic 6.0 würde die gleiche Klassendefinition wie folgt lauten:

```
Dim m_sName As String

Property Let Name(Wert As String)
  m_sName = Wert
End Property

Property Get Name() As String
  Name = m_sName
End Property
```

7. Die Erfahrung hat gezeigt, dass es besser ist, anstelle einer Zuweisung an den Namen der Eigenschaft den neuen *Return*-Befehl für die Zurückgabe eines Wertes zu benutzen.

Tipp
Tippen Sie im Programmeditor von Visual Studio .NET in einer Klassendefinition den *Property*-Befehl ein und drücken Sie die ↵-Taste. Der Programmeditor fügt automatisch den *Get*- und *Set*-Zweig ein. Achten Sie aber darauf, dass dabei kein Datentyp eingesetzt wird. Wird dies nicht nachgeholt, erhält die Eigenschaft den Datentyp *Object*.

4.15.4 Die Rolle des Konstruktors

Ein *Konstruktor* ist eine Funktion, die mit dem Instanzieren der Klasse aufgerufen wird und fester Bestandteil objektorientierter Programmiersprache ist. Ein Konstruktor, den es bei Visual Basic 6.0 noch nicht gab, ist eine sehr praktische Einrichtung, denn auf diese Weise lassen sich mit dem Instanzieren der Klasse bereits einzelne Eigenschaften mit Werten vorbelegen, was ansonsten nur umständlich möglich wäre. Der Konstruktor ist bei VB.NET eine Prozedur (sie soll keinen Wert zurückgeben) mit dem Namen *New*, die »irgendwo« in der Klasse enthalten sein muss (Klassen kommen auch ohne Konstruktor aus – sie erhalten dann aber einen Default-Konstruktor).

Code-Beispiel
Das folgende Beispiel ergänzt die Klasse *CSpieler* um einen Konstruktor, der dafür sorgt, dass die Eigenschaft *Name* mit dem Instanzieren der Klasse einen Wert erhält.

```
Class CSpieler
' Definition der Mitglieder
' ...
' Und jeeetzt ... der Konstruktooor

  Sub New (Name As String)
    m_sName = Name
  End Sub

End Class
```

Dank des Konstruktors, dem beliebig viele Argumente übergeben werden können, erhält die Eigenschaft *Name* mit dem Instanzieren der Klasse ihren Wert:

```
oTorwart = New CSpieler("O. Kahn")
WriteLine("Der Spieler heisst: " & oTorwart.Name)
```

4.15.5 Vererbung

Vorhang auf, hier kommt die Vererbung. Wohl kaum ein anderes Merkmal dürfte bei Visual Basic so lange so schmerzlich vermisst worden sein und wohl kaum ein anderes Merkmal präsentiert sich so unspektakulär. Es ist daher wieder einmal alles halb so wild. Die Vererbung ist ein Standardmerkmal praktisch aller Programmiersprachen und bewirkt natürlich keine Wunder. Vererbung bedeutet, dass bei der Definition einer neuen Klasse eine andere Klasse angegeben werden kann,

deren Mitglieder (entweder vollständig oder teilweise) in der neuen Klasse automatisch zur Verfügung stehen. Die neue Klasse »erbt« die bereits vorhandenen Mitglieder der so genannten *Basisklasse*, so dass die Programmierin diese in der neuen Klasse nicht erneut implementieren muss. Dabei geht es weniger um eine Arbeitsersparnis, sondern in erster Linie um ein konsistentes Programmdesign, bei dem eine Funktionalität nur einmal implementiert und anschließend mehrfach benutzt werden kann, wobei es jeder Klasse offen steht, die geerbte Funktionalität zu modifizieren, in dem z.B. einzelne Methoden überschrieben werden. Damit ist Vererbung ein wichtiges Merkmal moderner Programmiersprachen.

Vererbung wird bei VB.NET über den Befehl *Inherits* implementiert. Ihm muss der Name der Basisklasse folgen, von der die neue Klasse erben soll.

Code-Beispiel
Das folgende Beispiel erweitert das Beispiel aus dem letzten Abschnitt um eine neue Klasse *CTorwart,* die über den *Inherits*-Befehl alle Mitglieder der (Basis-) Klasse *CSpieler* erbt.

```
Imports System.Console

Module Modul1

Class CSpieler
  Dim m_sName As String
  Property Name() As String
   Get
    Name = m_sName
   End Get
   Set
    m_sName = Value
   End Set
  End Property

  Sub New (Name As String)
    m_sName = Name
 End Sub
End Class

Class CTorwart
  Inherits CSpieler
  Private m_iAnzahl As Integer
  Property ElfmeterGehalten() As Integer
   Get
     ElfmeterGehalten = m_iAnzahl
   End Get
   Set
     m_iAnzahl = Value
   End Set
  End Property

  Sub New (Name As String, Anzahl As Integer)
```

```
    MyBase.New (Name)
    m_iAnzahl = Anzahl
  End Sub
End Class

Sub Main ()
  Dim oTorwart As CTorwart = _
   New CTorwart("Dino Zoff", 34)
  WriteLine("Der Spieler heisst: " & oTorwart.Name)
  WriteLine("Anzahl gehaltener Elfmeter: " & _
   oTorwart.ElfmeterGehalten)
End Sub

End Module
```

Eine Besonderheit gibt es bezüglich des Konstruktors zu beachten. Da die neue Klasse von einer Klasse mit Konstruktor erbt, muss sie diesen Konstruktor in ihrem Konstruktor aufrufen, was über den Aufruf

```
MyBase.New (Name)
```

geschieht. Das Schlüsselwort *MyBase* steht dabei für die Basisklasse, von der die aktuelle Klasse abgeleitet wurde. Anschließend steht es ihr frei, den eigenen Konstruktor beliebig zu gestalten[8].

Wenn Sie das Programm (z.B. mit Notepad) erstellen, kompilieren und starten, wird der Name des Torwarts und die Anzahl seiner gehaltenen Elfmeter ausgegeben. Ein zugegeben recht unspektakuläres Beispiel, bei dem die Mechanik der Vererbung im Vordergrund steht. Es sei an dieser Stelle noch einmal erwähnt, dass die Vererbung von der .NET-Laufzeitumgebung zur Verfügung gestellt wird, und es z.B. kein Problem ist, dass eine Klasse in einem VB.NET-Programm von einer Klasse erbt, die in C#, C++ oder einer x-beliebigen anderen .NET-Programmiersprache implementiert wurde. Das macht die Vererbung zu einem sehr leistungsfähigen Element.

4.15.6 Weitere »tolle« OOP-Eigenschaften

Über die VB.NET-Klassen gibt es noch sehr viel mehr zu berichten. Neben der Vererbung gibt es mit der Möglichkeit, Methoden überschreiben und überladen zu können weitere, zwar weniger spektakuläre, aber dennoch wichtige Neuerungen. Auch die Schnittstellenprogrammierung, d.h. das Implementieren von abstrakten Basisklassen, die lediglich die Namen von Eigenschaften und Methoden, aber keinen Programmcode enthalten, und die bei Visual Basic 6.0 etwas »halbgar« über den *Implements*-Befehl realisiert wurde, ist bei VB.NET auf eine neue und erheblich erweiterte Grundlage gestellt worden. Mehr zur Klassenprogrammierung in Kapitel 5, in der es um eine vollständige Übersicht über alle neuen Möglichkeiten, Befehle und Schlüsselwörter geht.

8. Das ist zugegebenermaßen etwas unschön, denn der Aufruf der Basisklasse sollte implizit erfolgen.

4.16 Multithreading

Multithreading ist sicherlich das zweite große Thema neben der Vererbung, was die Neuerungen bei VB.NET betrifft. Bis Visual Basic 6.0 liefen Visual Basic-Programme immer nur in einem Thread. VB.NET bietet dagegen ein erstklassiges *Multithreading*. Dadurch können in einem VB.NET-Programm beliebige viele *Threads* (zu deutsch »Programmausführungsfäden«) gestartet werden, die man sich wie viele kleine Subprogramme vorstellen kann, die parallel zum Hauptprogramm (dem Hauptthread) ausgeführt werden. Durch den geschickten Einsatz von Threads lässt sich die Ausführungsgeschwindigkeit eines Programms deutlich steigern, da Leerlaufzeiten im Hauptthread (etwa das Warten auf eine Tastatureingabe) durch die Ausführung von Subthreads besser gefüllt werden können, was in einem Single-Thread-Programm, wie es bei allen Visual Basic 6.0-Programmen der Fall war, nicht möglich ist. Während der erste Thread auf die Eingaben des Benutzers wartet, kann ein zweiter Thread eine umfangreichere Berechnung durchführen und ein dritter Thread kann, sofern dies benötigt wird, eine Anfrage bei einem Webserver im Internet starten. Bei einem Multiprozessorensystem ist es sogar möglich, dass das Betriebssystem mehrere Threads auf mehrere Prozessoren verteilt und somit eine echte Parallelverarbeitung entsteht. Dafür, dass die Threads der Reihe nach dran kommen, sorgen das Betriebssystem und die dafür zuständige *Threading*-Klasse der .NET-Klassenbibliothek, die u.a. auch die notwendigen Synchronisationsobjekte zur Verfügung stellt.

Einige Visual Basic-Programmierer haben sich daher Multithreading bereits seit der Version 1.0 gewünscht. Ansatzweise wurde es schon früher verwirklicht, da es seit Windows NT 3.1 (der allerersten Version) und Windows 95 vom Betriebssystem unterstützt wird. Während Multithreading bei Version 5.0 ansatzweise und bei Version 6.0 aufgrund einiger interner Änderungen im Grunde nicht mehr möglich war, steht es mit VB.NET uneingeschränkt zur Verfügung. Es wird allerdings nicht über VB.NET-Befehle, sondern über die Unterklassen, Methoden und Eigenschaften der *Threading*-Klasse in der .NET-Klassenbibliothek realisiert. Das bedeutet auch, dass Multithreading in C# auf die exakt gleiche Weise funktioniert wie in VB.NET.

Bevor nun die große »Multithreading-Euphorie« ausbricht, ist ein warnender Hinweis angebracht. Drei Dinge gilt es (wie üblich) zu berücksichtigen:

- Multithreading ist auch bei VB.NET nicht einfach zu programmieren und gehört zu den fortgeschritteneren Programmiertechniken.
- Durch das Starten mehrerer Threads ergibt sich nicht automatisch eine Geschwindigkeitssteigerung. Ob sich Threads positiv auswirken, hängt sehr stark vom Programmaufbau ab. Es kann sogar sein, dass Threads (etwa, wenn sie sich gegenseitig blockieren) ein Programm verlangsamen.
- Durch die Verwendung von Threads stellen sich völlig neuartige Probleme (ein Thread kann jederzeit von einem anderen Thread unterbrochen werden),

deren Lösung sehr viel Erfahrung erfordert und deren Nicht-Beachtung zu Programmabstürzen führen kann, die u.U. erst auftreten, nachdem das Programm lange Zeit fehlerfrei lief.

Die Quintessenz ist: Multithreading gehört, wie die Vererbung, zu den »Power-Befehlen« von VB.NET, wobei unerfahrene Programmierer damit schnell fehlerhaften Programmcode, der schwierig zu debuggen ist, produzieren können.

4.16.1 Multithreading-Unterstützung in VB.NET

VB.NET unterstützt Multithreading auf mehreren Ebenen:

- Über das *Thread*-Objekt der .NET-Klasse *System.Threading* lassen sich neue Threads anlegen und wieder beenden. Die Adresse der als Thread auszuführenden Funktion oder Prozedur wird über den bekannten *AddressOf*-Operator übergeben.
- Über den *SynchLock*-Befehl wird eine Synchronisation mehrerer Threads bei der Ausführung ein- und desselben Programmbereichs bewirkt.
- Das ist für Programmierer sicherlich der wichtigste Aspekt: Die Visual Studio .NET-IDE unterstützt das Debuggen von Multithreading-Programmen.

4.16.2 Das Anlegen eines Threads

Das Anlegen eines Threads ist sehr einfach. Über *New* wird ein neues *Thread*-Objekt angelegt, dem über *AddressOf* die aufzurufende Prozedur übergeben wird. Letztere muss eine Methode einer Klasse sein.

> **Code-Beispiel**
> Das folgende Code-Beispiel soll im Rahmen einer kleinen Übung Schritt für Schritt umgesetzt werden. Dazu wird nicht Visual Studio .NET benötigt, es genügt, die Programmdatei mit Notepad zu erstellen und den Compiler *Vbc.exe* aufzurufen.

Schritt 1:

Legen Sie eine neue Konsolenanwendung an und geben Sie dem Projekt den Namen »HalloThread«.

Schritt 2:

Fügen Sie in das Modul als erstes einen *Imports*-Befehl ein. Dieser erweitert den Namensraum um jene *Threading*-Klasse, in der alle für den Umgang mit Threads enthaltenen Klassen zusammengefasst sind. Für die Ausgabe wird wie üblich die *Console*-Klasse benötigt.

```
Imports System.Threading
Imports System.Console
```

Schritt 3:

Definieren Sie eine neue Klasse. Diese dient lediglich dazu, eine Methode (in diesem Beispiel heißt sie »ThreadWorker«) zur Verfügung zu stellen, die später als eigener Thread aufgerufen werden kann:

```
' Dieser Thread zählt von 1 bis 50 und endet dann wieder
Class CThread
' Die Methode ThreadWorker wird als eigener Thread ausgeführt

  Public Sub ThreadWorker()
   Dim iCounter As Integer
   For iCounter = 1 To 20
     WriteLine("Meldung aus dem Tread: " & iCounter.ToString())
   Next iCounter
  End Sub

End Class
```

Die Methode *ThreadWorker* macht nichts anderes als die Zahlen 1 bis 20 auf den Bildschirm auszugeben.

Schritt 4:

Definieren Sie die *Main*-Prozedur des Modul.

```
' Hier beginnt die Programmausführung
' Dies ist der Hauptthread des Programms
Sub Main()
```

Schritt 5:

Die Aufgabe von *Main* ist es, ein neues *Thread*-Objekt anzulegen:

```
Dim iCounter As Integer
' Eine neue Instanz der Klasse mit dem Thread anlegen
Dim oMT As CThread = New CThread()

' Ein neues Thread-Objekt anlegen
' Die Thread-Klasse ist Teil des System.Threading-Namensraums
Dim oNewThread As Thread

' Anlegen eines neuen Threads
' Thread soll die Methode ThreadWorker ausführen
oNewThread = New Thread(New ThreadStart(AddressOf oMt.ThreadWorker))
```

Schritt 6:

Damit wurde ein neuer Thread angelegt, der im Folgenden auch gestartet werden soll (ansonsten passiert nichts):

```
' Der neue Thread wird gestartet
' Die Methode ThreadWorker läuft in einem anderen Thread
oNewThread.Start()
```

Schritt 7:

Damit sich feststellen lässt, dass ein weiterer Thread aktiv ist, soll auch der Hauptthread etwas arbeiten:

```
' Auch der Hauptthread soll etwas arbeiten
  For iCounter = 100 To 120
    WriteLine("Meldung aus dem Hauptprogramm: " & _
    iCounter.ToString())
  Next iCounter
  ReadLine()
 End Sub
End Class
```

Schritt 8:

Starten Sie das Programm über F5 oder kompilieren Sie die Datei und führen Sie sie aus. Wenn Sie alles richtig gemacht haben, sollten im Ausgabefenster »abwechselnd« die Meldungen der beiden Threads erscheinen. Beide Threads werden durch das Betriebssystem gesteuert gleichzeitig ausgeführt, wobei es aufgrund unterschiedlicher Prioritäten und anderer Details, die sich durch das Programm beeinflussen lassen (dann wird es aber relativ kompliziert) auch eine gleichberechtigte Ausführung ermöglichen sollte.

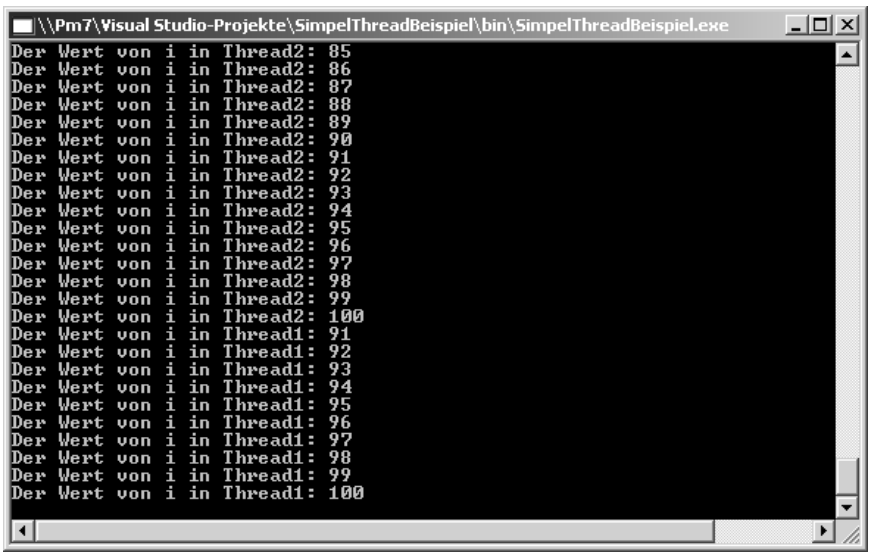

Abbildung 4.5: Die Ausgaben in dem Fenster stammen von zwei Threads

Ein Thread kann über die Methoden *Abort* oder *Stop* beendet werden. Über die *Sleep*-Methode wird der Thread für eine festgelegte Dauer angehalten.

Die *System.Threading*-Klasse stellt eine Vielzahl von Objekten bereit, die z.B. zur Synchronisation von Threads verwendet werden können, auf die in diesem Buch aber nicht eingegangen wird.

4.17 Delegates für Callback-Funktionen und andere Dinge

Und nun wieder zu etwas Neuem, zu dem es in Visual Basic 6.0 kein direktes Pendant gibt, den *Delegates*. Ein Delegate ist eine .NET-Klasse, die den Aufruf einer beliebigen Methode einer Klasse ermöglichen soll (im Prinzip auch den Aufruf einer Funktion in einem Modul, doch da Module bei VB.NET Klassen sind, läuft es auf dasselbe hinaus). Ein Delegate ist ein *typsicherer Funktionszeiger* und damit eine Möglichkeit, Funktionen über ihre Adresse aufzurufen, wobei vor dem Aufruf anhand der Signatur (d.h. den Datentypen der einzelnen Funktionsparameter) geprüft wird, ob der Aufruf überhaupt erlaubt ist. Was sich zunächst etwas exotisch anhören mag, aber relativ einfach zu verstehen ist. Soll eine beliebige Klasse *A* eine Methode einer anderen Klasse *B* aufrufen, übergibt man der Klasse *A* ein *Delegate*-Objekt, das die Adresse der Methode in der Klasse *B* enthält. Klasse *A* ruft die Funktion über das *Delegate*-Objekt auf, was sicherstellt, dass die Methode stets in der richtigen Weise aufgerufen wird, was bei reiner Übergabe einer Adresse nicht gewährleistet wäre. Der Name »Delegate« leitet sich (vermutlich) von dem Wort »Delegierter« ab. Man sendet einen Delegierten, der die Funktionsadresse und das Wissen über die Datentypen der aufzurufenden Funktion mit sich trägt und dafür sorgt, dass beim Aufruf nichts schief gehen kann. *Delegates* sind ein sehr wichtiges Konzept im .NET-Framework. Unter anderem basiert der gesamte Ereignismechanismus der Windows Forms-Formulare und ihrer Steuerelemente (siehe Kapitel 6) auf *Delegate*-Objekten.

Code-Beispiel

Das folgende Beispiel demonstriert in erster Linie die Mechanik beim Umgang mit *Delegate*-Objekten. Es besteht aus zwei (harmlosen) Prozeduren *P1* und *P2*, die über ein *Delegate*-Objekt mit dem Namen *DelegateNr1* aufgerufen werden sollen. Dabei wird für den Aufruf einmal die *Invoke*-Methode und einmal eine Abkürzung verwendet. Zum Schluss wird auch ein Beispiel für einen so genannten »Multicast«-*Delegate* gegeben. Dies ist ein *Delegate*, in dem über die *Combine*-Methode mehrere *Delegates*, die aber kompatibel bezüglich der Datentypen der aufrufenden Funktionen sein müssen, zusammengefasst werden. Es lassen sich über einen einzigen Aufruf mehrere Funktionen aufrufen.

Für die Umsetzung des Beispiels müssen Sie eine Konsolenanwendung anlegen und die folgenden Befehle eingeben (außerdem muss der neue Name des Moduls in den Projekteigenschaften ausgewählt werden).

```
Imports System
Module basSimpelDelegate
Delegate Sub DelegateNr1(ByVal sString As String)

Sub P1(ByVal Ausgabe As String)
    Console.WriteLine("Aus Sub P1: " & Ausgabe)
End Sub

Sub P2(ByVal Ausgabe As String)
    Console.WriteLine("Aus Sub P2: " & Ausgabe)
End Sub

Sub Main()
' Delegate D1 ruft die Prozedur P1 auf
    Dim D1 As New DelegateNr1(AddressOf P1)
    D1.Invoke("Hallo, Delegate Nr. 1")
' Delegate D2 ruft die Prozedur P2 auf
    Dim D2 As DelegateNr1
    D2 = New DelegateNr1(AddressOf P2)
    D2("Hallo, und nun Delegate Nr. 2")
    Console.ReadLine()
' Jetzt kommt ein Multicast-Delegate ins Spiel
    Dim D3 As DelegateNr1
    D3 = CType(System.Delegate.Combine(D1, D2), DelegateNr1)
    D3("Und nun beide auf einmal!")
    Console.ReadLine()
End Sub

End Module
```

Code-Beispiel

Das folgende Beispiel ist etwas anspruchsvoller. Es handelt sich um eine in einer Konsolenanwendung eingebettete *BubbleSort*-Sortierfunktion, die ein Zahlenfeld sortiert, wobei die eigentliche Vergleichsfunktion, die bestimmt, ob die beiden verglichenen Elemente ihre Position tauschen, über ein *Delegate*-Objekt übergeben wird.

```
Module basBubbleSort

' Als erstes wird eine Delegate-Funktion definiert
Delegate Function delVergleich(ByVal x As Integer, _
                    ByVal y As Integer) As Boolean

' Dies ist die Funktion mit einem Delegate aufgerufen wird, der den
eigentlichen Vergleich vornimmt

Function BubbleSort(ByVal VergleichFun As delVergleich,
 ByVal intZahlen() As Integer) As Integer()
  Dim i, j, Wert, Temp As Integer
' Alle Zahlen des Feldes der Reihe nach durchgehen
  For i = 0 To UBound(intZahlen)
    Wert = intZahlen(i)
```

```
' Den aktuellen Wert mit allen folgenden Werten des
' Feldes vergleichen
    For j = i + 1 To UBound(intZahlen)
' Den Vergleich führt die Delegate-Funktion durch
      If VergleichFun.Invoke(intZahlen(j), Wert) Then
' Delegate ergab Feldelemente müssen getauscht werden
        Temp = intZahlen(j)
        intZahlen(j) = Wert
        intZahlen(i) = Temp
        Wert = Temp
      End If
    Next j
  Next i
End Function

' Diese Funktion wird über den Delegate für den
' Vergleich in aufsteigender Reihenfolge aufgerufen
  Function VergleichAufFun (ByVal z1 As Integer, _
    ByVal z2 As Integer) As Boolean
    Return z2 > z1
  End Function

' Diese Funktion wird über den Delegate für den
' Vergleich in absteigender Reihenfolge aufgerufen
  Function VergleichAbFun(ByVal z1 As Integer, _
    ByVal z2 As Integer) As Boolean
    Return z2 < z1
  End Function

' Hier beginnt die Programmausführung
  Sub Main()
    Dim intZahlen() As Integer = _
     {77, 11, 23, 14, 99, 47, 67, 32}
    Dim i As Integer
' Ein neues Delegate-Objekt mit dem Namen Vergleich
' vom Typ delVergleich definieren
' Dim Vergleich As New delVergleich _
' (AddressOf VergleichAufFun)
  Dim Vergleich As New delVergleich _
   (AddressOf VergleichAbFun)
  Console.WriteLine("Das zu sortierende Zahlenfeld")
  For i = 0 To intZahlen.Length - 1
    Console.WriteLine("Der Wert an Position {0} ist {1} ", _
   i, intZahlen(i))
  Next i
  BubbleSort(Vergleich, intZahlen)
  Console.WriteLine("Zahlenfeld nach Bubblesort-Sortierung mit
Delegate:")
  For i = 0 To intZahlen.Length - 1
    Console.WriteLine("Der Wert an Position {0} ist {1} ", _
   i, intZahlen(i))
  Next i
```

```
    Console.ReadLine()
    End Sub
End Module
```

4.18 Strukturierte Ausnahmebehandlung

Zunächst vorweg, der gute alte *On Error*-Befehl bleibt uns auch bei VB.NET erhalten und es gibt nach wie vor keine »globale Fehlerbehandlung«. Dafür gibt es mit den Befehlen *Try*, *Catch* und *Finally* eine neue und in vielen Fällen sehr viel praktischere Art und Weise mit Laufzeitfehlern, sie werden in diesem Zusammenhang als Ausnahmen (engl. »exceptions«) bezeichnet, umzugehen. Diese Befehle werden unter dem Namen *strukturierte Ausnahmebehandlung* zusammengefasst. Der zunächst etwas akademisch klingende Name sollte zu keinen falschen Annahmen verleiten. Am Prinzip der Laufzeitfehlerbehandlung auf lokaler Ebene hat sich oberflächlich betrachtet zunächst nichts geändert, wenngleich es mit den mächtigen Klassen der .NET- Klassenbibliothek ganz neue und teilweise ungeahnte Möglichkeiten gibt, einem Laufzeitfehler, pardon, einer Ausnahme, bis ins kleinste Detail auf die Spur zu kommen.

Hinter einer *Exception*-Klasse steht eine kleine Hierarchie an Klassen, die für die Ausnahmebehandlung zuständig sind. Von der *Exception*-Klasse leiten sich die Klassen *ApplicationException* und *SystemException* ab. Während *Application Exception* durch die Anwendung (d.h. nicht durch die .NET- Klassenbibliothek) ausgelöst wird, wird eine *SystemException* durch die CLR immer dann ausgelöst, wenn in der Anwendung ein abfangbarer Laufzeitfehler aufgetreten ist.

Was sich zunächst relativ kompliziert anhören mag, ist zum Glück relativ einfach. Vor allem, und darin liegt der wichtigste Vorteil, ist ein solches Konstrukt etwas logischer aufgebaut. Anders als bei *On Error* bezieht sich die neue Laufzeitfehlerbehandlung nicht automatisch auf eine komplette Prozedur, sondern auf einen einzelnen Programmteil, wobei die Befehlswörter *Try*, *Catch*, *Finally* und gegebenenfalls *Throw* in der aufgeführten Reihenfolge zum Einsatz kommen.

Als erstes folgt auf *Try* der auszuführende Befehl.

Code-Beispiel

```
Try
  Dim oFi = File.New("IrgendeineDatei")
```

Sollte dieser Befehl fehlschlagen, werden automatisch die auf *Catch* folgenden Befehle ausgeführt.

Code-Beispiel

```
Catch
  LogError "Datei konnte nicht geöffnet werden!"
```

Dies ist der eigentliche Teil der Laufzeitfehlerbehandlung. Doch es geht noch weiter. Über *Finally* wird jener Befehlsblock angegeben, der in jedem Fall ausgeführt werden soll. Dieser Teil ist optional, d.h. wenn es nichts auszuführen gibt,

muss auch nichts ausgeführt werden. Schließlich gibt es über *Throw* die Möglichkeit, einen Laufzeitfehler auszulösen (etwa, um ihn an eine höhere Programmebene weiterzureichen – dies entspricht der *RaiseError*-Methode von Visual Basic 6.0).

Code-Beispiel
Hier noch einmal die komplette *Try-Catch-Finally*-Sequenz am Beispiel eines Dateizugriffs.

```
Sub Main()
  Dim sDateiName As String = "C:\Autoexec.batx"
  Dim sInhalt, sZeile As String
  Try
    Open(1, sDateiname, OpenMode.Input)
  Catch
    sDateiname = _
     Inputbox("Bitte Dateinamen korrigieren", , sDateiname)
     Open(1, sDateiname, OpenMode.Input)
  Finally
     ' Hier gibt es nichts zu tun
  End Try

    Do While Not EOF(1)
      sZeile = LineInput(1)
      sInhalt += sZeile & vbCrLf
    Loop
    Close(1)
    Msgbox(sInhalt)
End Sub
```

In diesem, zugegeben etwas konstruierten, Beispiel wird zunächst versucht, eine Datei mit einem nicht existierenden Dateinamen zu öffnen. Im *Catch*-Teil erhält der Anwender die Gelegenheit den Namen zu korrigieren.

4.18.1 Abfragen der Fehlerinformationen

Normalerweise interessiert man sich für die Ursachen eines Fehlers und möchte nach dem Motto »Was, wann, wer und warum« gewisse Details wissen. Diese Details liefert bei Visual Basic 6.0 das *Err*-Objekt, ohne allerdings allzu sehr ins Detail zu gehen. Bei VB.NET erhält man eine Fülle von Informationen über die Klassen *Exception* bzw. *SystemException*. Sie finden den *Catch*-Befehl häufig in der folgenden Varianten:

```
Catch oSysEx As SystemException
  Msbgox ("Der Fehler lautet: " & oSysEx.Message)
```

In diesem Beispiel wird die Laufzeitfehlerinformation über die lokale Variable *oSysEx* vom Typ *SystemException* zur Verfügung gestellt.

4.19 Dateizugriffsbefehle

Dies ist ein Bereich wo die Grenze zwischen der Programmiersprache VB.NET und der darunterliegenden .NET-Klassenbibliothek langsam verschwindet. Während bis Visual Basic 6.0 die Dateizugriffsbefehle ein fester Bestandteil der Programmiersprache waren, werden sie bei VB.NET (in leicht abgewandelter Form) nur noch aus Kompatibilitätsgründen über ähnlich lautende Methoden zur Verfügung gestellt. Dadurch ergibt sich eine geringfügig andere Schreibweise, wobei sich am Prinzip des Zugriffs nichts geändert hat. Die neuen »Dateizugriffsbefehle« werden über die *System.IO*-Klasse zur Verfügung gestellt, die (wen wunderts?) eine Vielzahl von Möglichkeiten bietet.

Code-Beispiel
Das folgende Beispiel liest mit Hilfe der in der Kompatibilitätsklasse *Microsoft.VisualBasic* enthaltenen Methoden den Inhalt der Datei *Autoexec.bat* aus (sofern vorhanden, ansonsten ist ein Laufzeitfehler die Folge, der nicht abgefangen wird) und zeigt ihn in einer Mitteilungsbox an.

```
Imports System
Imports Microsoft.VisualBasic

Module Modul1

Sub Main ()
  Dim sDateiname As String = "C:\Autoexec.bat"
  Dim sInhalt As String
  Dim iKanalNr As Integer = FreeFile()
  FileOpen (iKanalNr, sDateiname, OpenMode.Input)
    sInhalt = InputString(iKanalNr, CInt(LOF(iKanalNr)))
  FileClose()
  Msgbox (sInhalt)
End Sub
End Module
```

Wie das kleine Beispiel zeigt, sind bei VB.NET die Dateizugriffe irgendwie vertraut, aber doch irgendwie ein wenig anders. Besonders pingelig verhält sich VB.NET bei den ansonsten harmlosen Konstanten, die etwa den Dateizugriffsmodus bestimmen. Hier muss die Zahl vom Typ einer Enumeration sein, die in der Kompatibilitätsklasse enthalten ist. Eine weitere Besonderheit ist, dass die *LOF*-Funktion offenbar einen *Long*-Wert zurückgibt, der entsprechende Parameter, der die Anzahl der zu lesenden Zeichen festgelegt, aber vom Typ *Integer* sein muss.

Hinweis
Wie alle COM-Bibliotheken lässt sich auch die *Microsoft Scripting-Runtime (Scrrun.dll)* mit dem beliebten *FileSystemObject*-Objekt in einem VB.NET-Programm ansprechen. Die *System.IO*-Klasse ersetzt mit ihren Klassen *File* und *Directory* jedoch fast die gesamte Funktionalität, so dass die Scripting Runtime für neue Projekte nicht mehr benutzt werden sollte.

Code-Beispiel
Der folgende Code-Ausschnitt zeigt, wie sich mit der *Directory*-Klasse die Anzahl der Dateien und Unterverzeichnisse in einem Verzeichnis ausgeben lässt:

```
Private oDir As Directory
WriteLine(oDir.GetFiles("C:\").Length & " Objekte in C:\")
```

Code-Beispiel
Das folgende Beispiel zeigt, wie sich der Inhalt einer Textdatei über die *System.IO*-Klasse und das darin enthaltene *StreamReader*-Objekt einlesen lässt.

```
Imports System.IO
Imports Microsoft.VisualBasic

Module Hauptmodul

  Sub Main()
    Dim sDateiname As String = "C:\Autoexec.bat"
    Dim oSt As StreamReader = New StreamReader(sDateiname)
    Msgbox(oSt.ReadToEnd())
    oSt.Close()
  End Sub

End Module
```

Code-Beispiel
Auch das nächste Beispiel zeigt, wie sich der Inhalt einer Textdatei einfach einlesen lässt. Dieses Mal kommt die *File*-Klasse zum Einsatz, deren *OpenText*-Methode ein *StreamReader*-Objekt zurückgibt:

```
Imports System.Console
Imports System.IO

Module basFileBeispiel
  Private oFi As File
  Private oSt As StreamReader
  Sub Main()
    oSt = oFi.OpenText("C:\Autoexec.bat")
    WriteLine(oSt.ReadToEnd.ToString)
    oSt.Close()
    ReadLine()
  End Sub

End Module
```

Diese Variante ist deutlich kürzer, kompakter und ganz im Stile von .NET. So verhält es sich in vielen Bereichen. Alt bewährte Visual Basic-Programmiertechniken lassen sich zwar übertragen, es ist aber sehr viel einfacher und eleganter, sie mit den Möglichkeiten der .NET-Klassen neu zu programmieren.

4.20 Befehle, die es »leider« nicht mehr gibt

Von einer Reihe von Befehlen heißt es beim Wechsel zu VB.NET leider Abschied nehmen. Allzu schwer wird dieser Abschied (falls es überhaupt jemanden auffallen sollte) aber nicht werden, denn die obsoleten Befehle fallen eher in die Kategorie »Ich wusste gar nicht, dass es die noch gibt«. Tabelle 4.3 gibt eine Übersicht.

Tabelle 4.3: Befehle, die es bei VB.NET nicht mehr gibt

Befehl	Mögliche Alternative
LSet	Keine[a]
Let	Keine
Set	Keine
Option Base	Keine
On n Goto, GoSub	»Normale Programmiermethoden«, d.h. Prozeduren, Entscheidungen usw.
GoSub/Return	Prozeduren
Def<Datentyp>	Normale Deklarationsbefehle

a. Das soll nicht heißen, dass es gar keine Alternative gibt. Es soll in erster Linie heißen, dass es sich vermutlich nicht lohnt, über eine solche nachzudenken.

4.20.1 Der LSet-Befehl

Dieser Befehl tauchte hin und wieder in recht obskuren Zusammenhängen auf (etwa beim Zugriff auf Random-Dateien, wenn es darum ging, einen eingelesenen Datensatz linksbündig auszurichten). Außerdem war er angeblich dazu da, einer Variablen mit einem benutzerdefinierten Datentyp den Wert einer Variablen des gleichen Typs zuzuweisen, was aber auch ohne *LSet* funktioniert. Kaum ein Visual Basic-Programmierer dürfte diesen Befehl daher wirklich vermissen.

4.20.2 Der Let-Befehl

Dies ist der Klassiker unter den Basic-Befehlen, den es in allen Visual Basic-Versionen gegeben hat, der aber vermutlich von niemanden benutzt wurde (den Basic-Fans der ersten Stunde einmal abgesehen[9]). Sie können den Befehl gerne in den Visual Studio .NET-Editor eintippen und werden gar nicht so schnell schauen können wie er (genau wie sein obsolet gewordener *Set*-Kollege) wieder verschwindet. Direkt im Quellcode wird er allerdings nicht unterstützt und erzeugt eine Fehlermeldung beim Kompilieren.

9. Er löst daher gewisse nostalgische Erinnerungen aus, denn Let A=1 ist der erste Basic-Befehl, den ich jemals über eine Computertastatur eingetippt habe.

4.20.3 Der Set-Befehl

Der *Set*-Befehl wird für die Zuweisung einer Objektreferenz an eine Variable nicht mehr benötigt, da es bei VB.NET keine mehrdeutigen Zusammenhänge mehr geben kann. In Visual Basic 6.0 war die Befehlsfolge:

```
Dim oText As Variant
oText = Text1
```

nicht erlaubt, da es für Visual Basic nicht erkennbar war, ob die Variable *oText* eine Referenz auf das Textfeld *Text1* (in diesem Fall mit *Set*) oder den Wert der Default-Eigenschaft *Text* (in diesem Fall ohne *Set*) erhalten soll. Da es bei VB.NET (in den meisten Situationen) keine Default-Eigenschaft mehr gibt, wird auch der *Set*-Befehl als »Schiedsrichter«nicht mehr benötigt.

4.20.4 Der Option Base-Befehl

Dieser Befehl wird nicht mehr benötigt, da ein Array immer bei 0 beginnt (zumindestens in der Beta-2).

4.20.5 Die Befehle GoSub und Return

In einer Prozedur können keine »Unterprogramme« mehr durch eine Sprungmarke definiert und aufgerufen werden. Bitte einmal melden, wer hat einen dieser Befehle in den letzten 5 Jahren benutzt? Niemand, gut dann können wir diese Befehle wohl streichen. So könnte es bei einem Entwicklermeeting zugegangen sein, bei dem es um die Frage ging, welche Befehle übernommen werden und welche nicht.

4.20.6 Der Befehl On <Variable> GoSub

Einmal ehrlich, wer hat gewusst, dass es diesen Befehl bis Visual Basic 6.0 gab? Nur zur Erinnerung, unter Visual Basic 6.0 war folgende Konstruktion möglich:

```
Dim nWert As Long
nWert = 2
On nWert GoSub M1, M2, M3
Exit Sub
M1:
  MsgBox "nWert ist 1"
  Return
M2:
  MsgBox "nWert ist 2"
  Return
M3:
  MsgBox "nWert ist 3"
  Return
```

In Abhängigkeit des Wertes von *nWert* wurde eines der folgenden Unterprogramme angesprungen (alternativ hätte man auch *Goto* schreiben können). Das geht unter VB.NET nun nicht mehr.

4.20.7 Die Def<Datentyp>-Befehle

Diese Befehle waren praktisch, wenn man es von (sehr) alten Basic-Versionen her so gewohnt war und man wusste, dass sie existieren. Durch einen *Def<Datentyp>*-Befehl (<Datentyp> steht für einen Visual Basic-Datentyp) lässt sich bei Visual Basic 6.0 die Definition von Variablen etwas vereinfachen:

```
DefInt A-B

Private Sub Form_Load()
  Dim A
  A = 1.2345
  MsgBox "Datentyp von A: " & TypeName(A)
End Sub
```

Durch den *DefInt*-Befehl erhält die Variable *A* automatisch den Datentyp *Integer* (und nicht *Variant* bzw. *Double*), auch wenn dieser bei der Deklaration nicht angegeben wurde (eigentlich gar nicht so unpraktisch).

Zeichenketten fester Länge

Zeichenketten fester Länge gibt es bei VB.NET offiziell nicht mehr. Sie lassen sich über den Umweg der Visual Basic-Kompatiblitätsklassen aber trotzdem deklarieren, was aber (wie üblich) einen gewissen Geschwindigkeitsnachteil bedeuten dürfte:

Code-Beispiel
Das folgende Beispiel setzt voraus, dass über PROJEKT|REFERENZ HINZUFÜGEN ein Verweis auf die *Microsoft.VisualBasic.Compatibility*-Klasse hinzugefügt wurde.

```
Dim sFest10 As New FixedLengthString(10)
sFest10.Value = "012345789"
MsgBox(sFest10.Value)
```

Innerhalb von Strukturen sind Zeichenketten fester Länge nicht mehr erlaubt.

4.20.8 Die »undokumentierten« Funktionen ObjPtr, StrPtr und VarPtr

Einige »Optimierungstechniken«, wie der direkte Zugriff auf Zeichenketten über die *StrPtr*-Funktion, oder der direkte Speicherzugriff auf Variablen, lassen sich nicht mehr bzw. nur auf einem Umweg durchführen.

4.20.9 Statische Prozeduren und Variablen

Ging in Visual Basic 6.0 einer Prozedur das Schlüsselwort *Static* voraus, wurden dadurch alle Variablen, die in der Prozedur deklariert wurden, zu statischen Variablen. Bei VB.NET wird dies nicht mehr unterstützt. Statische Variablen müssen durch Variablen mit einem Gültigkeitsbereich auf Modulebene ersetzt (oder mit dem *Static*-Schlüsselwort der Kompatibilitätsklasse definiert) werden.

4.20.10 Missing und die IsMissing-Funktion

Den Spezialwert *Missing* (den ein optionale Prozedurparameter vom Typ *Variant* in einigen Situationen indirekt annahm) und dementsprechend die *IsMissing*-Funktion, gibt es bei VB.NET nicht mehr.

Code-Beispiel

```
Option Explicit

Private Sub Form_Load()

  Dim Nix
  P1 Nix
End Sub

Sub P1(Arg1 As Variant, Optional Arg2 As Variant)
  If IsMissing(Arg2) = True Then
     MsgBox "Da fehlt wohl was"
     MsgBox "Der Wert von P1: " & TypeName(Arg2)
  Else
  MsgBox "Der Wert von P1: " & TypeName(Arg2)
  End If
End Sub
```

Enthält eine *Variant*-Variable den Spezialwert *Error*, steht ihr (Default-) Wert für die Fehlermeldung (z.B. Fehler 448).

4.20.11 Der Umgang mit Null

Der direkte Umgang mit dem Spezialwert *Null* ist bei VB.NET nicht mehr möglich. Das *Null*-Schlüsselwort wurde durch *DBNull* ersetzt, was sich nur im Zusammenhang mit Datenbankfeldern einsetzen lässt. Die *IsNull*-Funktion heißt nun *IsDBNull*.

Code-Beispiel

Der folgende beliebte Trick, der die direkte Abfrage auf *Null* ersparte, ist bei VB.NET nicht mehr möglich.

```
txtName.Text = "" & adors.Fields("Name").Value
```

4.20.12 TypeName-Funktion bei nicht initialisierten Strings

Da Strings bei VB.NET Objekte sind, gibt die *TypeName*-Funktion bei nicht initialisierten Strings den Wert *Nothing* zurück und nicht mehr *String* wie früher.

4.21 Aufruf von API-Funktionen

Zunächst vorweg: Die Win32-API wird es auch in den nächsten Jahren geben[10] und auch wenn die .NET-Klassenbibliothek viele API-Funktionen »überflüssig« macht, wird es auch bei VB.NET-Programmen stets eine Notwendigkeit geben, die guten alten API-Funktionen aufzurufen[11]. Dem steht auch nichts im Wege. Im Gegenteil, der Aufruf ist sogar ein wenig einfacherer und sicherer (es gibt keine *As Any*-Deklaration mehr, die die Übergabe falscher Datentypen provozieren konnte) geworden.

Code-Beispiel
Das folgende Beispiel ist bewusst sehr einfach gehalten, indem es die API-Funktion *Beep* aufruft (die Funktion gibt einen Ton mit vorgegebener Frequenz und Dauer über den PC-Lautsprecher aus), der lediglich zwei Parameter übergeben werden.

```
Public Shared Sub <DllImport("kernel32.dll")> Beep _
 (ByVal freq As Integer, ByVal duration As Integer)
```

Bei der Deklaration fällt auf, dass es keinen speziellen Befehl (wie *Declare* bei Visual Basic 6.0) gibt, sondern die Funktion zunächst wie eine normale Funktion oder Prozedur definiert wird. Lediglich das Attribut *DllImport*, dem der Name der DLL-Datei folgt, teilt VB.NET mit, dass es sich hier um eine externe Funktion handelt. Der Aufruf selber ist völlig unspektakulär, denn die API-Funktion wird (genau wie bei Visual Basic 6.0) wie eine normale Funktion aufgerufen:

```
Beep(CInt(txtFrequenz.Text), CInt(txtDauer.text))
```

Hinweis
Es lohnt sich nicht, alle API-Aufrufe zu übernehmen, da die .NET-Klassen viele Aufgaben einfacher erledigen. Dazu gehört z.B. der Zugriff auf die Registrierung, der bei Visual Basic 6.0 noch eine Aneinanderreihung verschiedener API-Funktionen mit vielen Parametern erforderte. Bei VB.NET gibt es dafür die *Registry*-Klasse im *Microsoft.Win32*-Namensraum.

10. Das ist eine eher konservative Schätzung. Wie wäre es mit »ewig«?
11. So gibt es beispielsweise derzeit noch keine Klassen in .NET für das RAS-API.

4.21.1 Besonderheiten beim Aufruf von API-Funktionen

Folgendes muss beim Aufruf einer API-Funktion berücksichtigt werden:

- Die Größe des *Integer*- und *Long*-Datentyps haben sich geändert.
- Die Deklaration mit *As Any* ist nicht mehr möglich. Falls ein API-Aufruf mit verschiedenen Datentypen deklariert werden soll, muss die Deklaration mehrfach erfolgen.
- Erwartet eine API-Funktion einen Leerstring als Argument (wie es z.B. bei *FindWindow* oder *GetPrivateProfileString* der Fall ist), kann dieser mit einer *String*-Variablen übergeben werden, die auf *Nothing* gesetzt wurde.

4.22 Zusammenspiel mit der .NET-Klassenbibliothek

Die .NET-Klassenbibliothek ist die neue »API«, die Tausende von Klassen mit jeweils Dutzenden von Eigenschaften und Methoden bereit stellt, die ausnahmslos von einem VB.NET-Programm aufgerufen werden können. Auf welche Weise ein VB.NET-Programm diese Klassen benutzt, wurde in diesem Kapitel schon mehrfach vorgeführt. Es ist alles sehr unspektakulär, denn alles was getan werden muss ist, die Klassen zu instanzieren oder, falls es sich um ein statisches (*shared*) Mitglied handelt, die Eigenschaft oder Methode direkt aufzurufen. Irgendwelche Vorbereitungen oder Deklarierungen sind nicht erforderlich.

Code-Beispiel
Das folgende Beispiel zeigt eine etwas umständliche Art, Zufallszahlen zu erzeugen, die, wenn es um »richtige« Zufallszahlen geht, aber deutlich mehr Möglichkeiten bietet.

```
Dim oRnd As System.Random = New System.Random()
Msgbox(oRnd.Next)
```

Als erstes wird die .NET-Klasse *Random* instanziert (dass es diese Klasse überhaupt gibt und welche Mitglieder sie zur Verfügung stellt, erfährt man wie üblich a) durch Studium oder .NET-Klassenreferenz oder b) per Zufall). Anschließend wird über die *Next*-Methode die nächste Zufallszahl ausgegeben. Die .NET-Klassen sind daher eine natürlich Erweiterung (man könnte sie auch als festen Bestandteil sehen, da es keine Trennlinie zwischen Programmiersprache und Klassenbibliothek gibt) von VB.NET.

4.22.1 Die Rolle des Imports-Befehl

Praktisch jedes VB.NET-Programm beginnt mit einer Reihe von *Imports*-Befehlen. Diese sind weit aus weniger wichtig, als es vielleicht zunächst den Anschein haben mag. Sie sorgen lediglich dafür, dass sich die Unterklassen und Mitglieder der .NET-Klassen ohne Voranstellen des kompletten Pfadnamens ansprechen lassen (sie ähneln damit ein wenig dem Einfügen eines Verweises bei Visual Basic 6.0).

Code-Beispiel
Das folgende Beispiel, das bereits zu Beginn des Kapitels aufgeführt wurde, zeigt, wie sich durch Importieren der Referenz auf die *System.Console*-Klasse deren *WriteLine*-Methode direkt aufrufen lässt.

```
Imports System.Console
Module Hauptmodul
 Sub Main ()
  Writeline("Die Eulersche Kreiszahl ist etwa " & exp(1))
End Sub

End Module
```

Kurze Quizfrage zur Verständnisüberprüfung. Wird das obige Programm fehlerfrei kompilieren? Denken Sie bitte kurz nach. Die Antwort lautet: Nein, denn die *exp*-Funktion (sie berechnet den Logarithmus zur Basis 2) ist solange nicht bekannt, wie nicht a) der Name der *Math*-Bibliothek vorangestellt oder b) der Befehl *Imports.System.Math* an den Anfang gestellt wird (oder c) dies in den Projekteigenschaften eingestellt wird). Wichtig ist, dass der *Imports*-Befehl außerhalb der Moduldefinition aufgeführt werden muss.

4.22.2 Die Environment-Klasse

Als eine Klassen von vielen soll in diesem Abschnitt die *Environment*-Klasse vorgestellt werden, denn dank ihrer Mitwirkung lassen sich wichtige Informationen über die Umgebung, wie z. B. das aktuelle Verzeichnis, in dem das Programm läuft, die Betriebssystemversion oder die Version der .NET-Laufzeitumgebung, abfragen.

Code-Beispiel
Das folgende Beispiel überträgt den Pfad des Verzeichnisses, in dem die Anwendung ausgeführt wird, in eine Variable.

```
sPfad = Environment.CurrentDirectory.ToString)
```

Code-Beispiel
Das folgende Beispiel überträgt über die *GetFolder*-Methode den Verzeichnispfad des Verlauf-Ordners in eine Variable:

```
sPfad = Environment.GetFolderPath(Environment.SpecialFolder.History))
```

Frage: Wie lässt sich der obige Befehl ein wenig abkürzen? Antwort: Indem das Modul mit einem *Imports*-Befehl eingeleitet wird:

```
Imports System.Environment
```

Jetzt kann der Klassenname *Environment* entfallen, denn dieser ist dem Programm bekannt:

```
sPfad = GetFolderPath(SpecialFolder.History))
```

Die *Environment*-Klasse sollte als ein Beispiel von Hunderten gesehen werden, wie sich über die Klassenbibliothek zusätzliche Funktionen ansprechen lassen, die »früher« über API-Funktionen oder externe COM-Bibliotheken importiert werden mussten.

Tabelle 4.4: Interessante Eigenschaften der Environment-Klasse

Eigenschaft	Bedeutung
CommandLine	Die übergebenen Kommandozeilenparameter
CurrentDirectory	Das aktuelle Programmverzeichnis
ExitCode	Legt eine Zahl fest, die nach Beenden des Programms abgefragt werden kann
MachineName	Der Name des Computers
OsVersion	Gibt Informationen über die Betriebssystemversion zurück
UserName	Name des angemeldeten Benutzers
Version	Versionsnummer der .NET-Laufzeitumgebung

Tabelle 4.5: Interessante Methoden der Environment-Klasse

Methode	Bedeutung
Exit	Beendet das Programm und gibt den über die *ExitCode*-Eigenschaft festgelegten Wert an das Betriebssystem zurück
GetEnvironmentVariable	Gibt den Wert der angegebenen Umgebungsvariable zurück
GetFolderPath	Gibt den Verzeichnispfad eines über eine Konstante ausgewählten Systemordners zurück

4.23 Zugriff auf COM-Komponenten

COM-Komponenten sind für Visual Basic-Programmierer (nach anfänglicher Zurückhaltung) sehr wichtig geworden. Sei es, weil es selbst geschriebene Komponenten sind, in denen Teilfunktionalität einer Anwendung ausgelagert wird, sei es, weil es sich um vorkonfektionierte Windows-Funktionalität, wie z.B. bei der Microsoft Scripting Runtime, handelt. Um es kurz zu machen, das Einbinden von COM-Komponenten ist bei VB.NET kein Problem (wie es in Kapitel 2 erwähnt wurde, hat man auf die Interoperabilität besonderen Wert gelegt). Um z.B. das

FileSystemObject der Scripting Runtime einbinden zu können, sind in Visual Studio .NET folgende Schritte notwendig:

Schritt 1:

Ausführen des Menübefehls *Projekt|Verweis hinzufügen* und Auswahl des Registers *COM* (es dauert eine Weile bis alle Verweise aus der Registrierung eingelesen werden).

Schritt 2:

Auswahl der COM-Bibliothek, in diesem Fall der Microsoft Scripting Runtime, und Anklicken von *Auswählen*.

Schritt 3:

Bestätigen der Auswahl mit *OK*.

Anschließend kann die Scripting-Komponente »ganz normal« angesprochen werden.

Code-Beispiel

```
Dim oFso As New Scripting.FileSystemObject
Msgbox("Anzahl Verzeichnisse auf C:\ " &
oFso.GetFolder("C:\").SubFolders.Count)
```

Deklaration und Instanzierung erfolgen in einem Schritt. Würde die Instanzierung in einem separaten Schritt erfolgen, wäre kein *Set*-Befehl erlaubt – auch wenn es hier um ein gutes altes COM-Objekt geht.

4.24 Sonstige Neuerungen

In diesem Abschnitt geht es um weniger spektakuläre Änderungen, die aber natürlich nicht unerwähnt bleiben sollen.

4.24.1 Beim Prozeduraufruf sind Klammern Pflicht

Dieser Umstand hat gerade angehenden Visual Basic-Programmierern am Anfang erhebliche Verständnisschwierigkeiten bereitet. Vor allem die Frage, wann Klammern beim Aufruf einer Prozedur/Funktion gesetzt werden müssen und wann nicht. Bei VB.NET wurde dieser, im Grunde eher trivialer Umstand, vereinheitlicht. Das bedeutet aber auch, dass beim Aufruf der *MsgBox*-Funktion immer Klammern aufgeführt werden müssen. Eingeben müssen Sie diese Klammern zumindestens beim Visual Studio .NET-Editor nicht, denn sie werden netterweise automatisch eingefügt.

4.24.2 Kommentarzeilen über mehrere Zeilen

Kommentare werden auch bei VB.NET durch ein Apostroph (oder den *REM*-Befehl) eingeleitet und beschränken sich auf eine Zeile. Die Möglichkeit, wie in C# und praktisch allen anderen Programmiersprachen, Kommentarzeilen über mehrere Befehlszeilen ausdehnen zu können, soll erst mit dem Nachfolger von »VB.NET 1.0« nachgereicht werden.

4.24.3 Der Umgang mit Datumswerten

Datumswerte vom Typ *Date* werden intern nicht mehr als *Double*-Werte gespeichert, was aber nur in den seltensten Fällen eine Rolle spielen dürfte. Die Funktionen *Date* und *Time* werden durch *Today* und *TimeOfDay* ersetzt.

4.25 Zusammenfassung

Dieses Kapitel soll nicht mit einer Aufzählung mehr oder weniger belangloser Details enden (was tut man nicht alles der Vollständigkeit zu Liebe), sondern mit einer kurzen Zusammenfassung. Wie man zu den Neuerungen bei VB.NET steht, hängt sehr stark davon ab, wie weit man sich bereits mit dem Thema .NET beschäftigt hat (und natürlich von Temperament, Geschlecht, Alter und durchschnittlichem Jahreseinkommen). Als erfahrener Visual Basic-Veteran, der wieder einmal aus der Ferne hört, das bald alles anders wird, und dass (wieder einmal) alles auf Klassen basiert und man seinen Quellcode am besten wegwerfen soll, ist man versucht zunächst *Nein, danke* zu sagen. Zumal bislang alles wunderbar funktioniert hat. Hinter VB.NET, und das hat dieses Buch bisher hoffentlich halbwegs plausibel und glaubhaft deutlich machen können, steckt sehr, sehr viel mehr als ein neues Update, bei dem auch gleich alle Namen ausgetauscht wurden. .NET ist eine umfassende »Revolution« was die Betrachtungsweise und vor allem aber die Umsetzung der Softwareentwicklung von »Büro-Anwendungen« unter Windows angeht, bei denen die Anbindung an das Internet eine zentrale Rolle spielt. Statt wie bisher mit einem mehr oder weniger optimalen, aber letztendlich isolierten Werkzeug wie Visual Basic zu arbeiten und mehr oder weniger datentypenverträglich gegen verschiedene APIs und COM-Bibliotheken zu programmieren, gibt es eine große einheitliche Klassenbibliothek, die etwa 80% der benötigten Funktionalität enthält. Anstatt Bibliotheken einzubinden oder Funktionen aufzurufen, sind die Klassen der Bibliothek von Anfang an ein fester Bestandteil der Sprache. Welcher Sprache? Das spielt keine Rolle, denn bei .NET rücken sämtliche Programmiersprachen auf der (virtuellen) Rennstrecke auf die gleiche Startposition vor und werden mit den gleichen Motoren ausgestattet. Jetzt kommt es nur noch darauf an, wer der bessere Fahrer (sprich der bessere Programmierer) ist. Und das ist keine schlechte Ausgangsposition. Die Frage »Wer soll das alles bezahlen?« (sprich, wie sieht es mit der Portierbarkeit meiner Programme aus) muss unter diesem Blickwinkel gestellt werden. Zum einen muss nicht jede Visual Basic 6.0-Anwendung zwangsweise portiert werden, denn sie

wird auch unter einem »Windows 2005« aller Voraussicht nach problemlos laufen und das Einbinden sämtlicher COM-Komponenten sollte ebenfalls problemlos möglich sein. Zum anderen kostet der Fortschritt stets einen gewissen Preis, wobei es sich dieses Mal eher um eine Art Kredit handelt, denn mit dem gewonnenen .NET-Know-How lässt sich sehr viel mehr erreichen, als es in der Vergangenheit mit den doch oft eher etwas begrenzten Mitteln möglich war.