PEARSON Education

ADDISON-WESLEY

MASTER CLASS

Alex Bierhaus, Jürgen Kotz

Visual Basic 2005

Einstieg für Anspruchsvolle

Auf CD: Visual Basic 2005 Express Edition

ADDISON-WESLEY [in Kooperation mit] PEARSON Studium

6

Neue Features in Visual Basic 2005

In diesem Kapitel möchte ich gerne die wichtigsten Neuigkeiten der Version Visual Basic 2005 vorstellen.

6.1 Generics

Und beginnen will ich gleich mit dem Highlight aus meiner Sicht – **Generics**.

Sollten Sie in der Vergangenheit bereits viel mit **Collection-Objekten** gearbeitet haben, ist doch immer und immer wieder dasselbe Problem aufgetreten. Typensicherheit der Objekte innerhalb der Collection war nicht gegeben, da eine Collection standardmäßig eine Auflistung von Objekten, egal welchen Typs, ist.

Also machte man sich zumeist selbst an die Arbeit und hat mit mehr oder weniger Aufwand eigene typsichere Auflistungsobjekte geschrieben. Der Aufwand bei großen Objektmodellen war jedoch gewaltig, musste man doch für jeden unterschiedlichen Typ eine eigene Collection-Klasse schreiben.

Als zweites Problem kommt hinzu, dass bei der Verwendung von eigens entwickelten typsicheren Klassen intern immer zeitaufwändige Typumwandlungen stattfinden, um das Objekt auch mit dem korrekten Datentyp zurückgeben zu können. Mit diesen Typumwandlungen mussten auch alle Entwickler kämpfen, die sich den Aufwand nicht machen wollten, typsicherer Collections zu entwickeln. Jeder Zugriff auf die `Items`-Methode musste mit `CType` versehen werden, um auch den gewünschten Typ zu erhalten.

Mit .NET 2.0 führt Microsoft ein neues Konzept ein, das in etwa mit den Templates in C vergleichbar ist und alle oben beschriebenen Probleme löst. Generische Auflistungsobjekte, `Generics`, wurden in einem eigenen

Namensraum `System.Collections.Generic` bereitgestellt. Die bisher eingeführten Collection-Objekte sind weiterhin völlig abwärtskompatibel in `System.Collections` vorhanden.

Was bieten uns nun aber diese Generic-Klassen? Und die gute Nachricht ist: Sie bieten all das, was wir uns seit Jahren wünschen. Beim Konstruktoraufruf müssen wir lediglich den Typ angeben, für den die Auflistung verwendet werden soll, bei Schlüssel-Werte-Paar-Auflistungen können wir auch den Typ des Schlüssels angeben, und schon ist diese Auflistung typensicher. Es können nur noch Objekte von diesem speziellen Typ hinzugefügt werden, und auch die Rückgabewerte aus diesem speziellen Auflistungsobjekt sind nicht mehr vom Typ `Object`, sondern genau von diesem speziell angegebenen Typ. Dabei finden auch intern keine zeitaufwändigen Typumwandlungen statt, denn diese Klassen werden generisch erzeugt und sind vollkommen compilerunterstützt. Sogar **IntelliSense** funktioniert an den Stellen, wo Objekte dieses Typs zurückgegeben werden.

Hinweis

Eine Collection mit einer Schlüssel-Werte-Paar-Beziehung ist eine Auflistung, in der zu jedem Objekt zusätzlich ein eindeutiger Schlüsselwert gespeichert wird. Über diesen Schlüssel kann man dann sehr schnell auf das gewünschte Objekt zugreifen, ohne die Auflistung sequenziell durchlaufen zu müssen.

Innerhalb des Namensraums `System.Collections.Generic` gibt es eine ganze Reihe von unterschiedlichen generischen Auflistungsobjekten, deren wichtigste ich in Tabelle 6.1 kurz aufliste.

Tabelle 6.1 Klassen im Namespace System.Collections.Generic

Klasse	Beschreibung
Dictionary	Stellt ein Auflistungsobjekt mit einer Schlüssel-Werte-Paar-Beziehung dar.
LinkedList	Stellt eine doppelt verknüpfte Liste dar.
List	Stellt ein Auflistungsobjekt dar, über das über den Index auf die Objekte zugegriffen werden kann.
Queue	Stellt ein First-In-First-Out(FIFO)-Auflistungsobjekt dar.
SortedDictionary	Stellt ein Auflistungsobjekt mit einer Schlüssel-Werte-Paar-Beziehung dar, die nach dem Schlüsselbegriff sortiert ist.
SortedList	Stellt ein Schlüssel-Werte-Paar-Auflistungsobjekt dar, das aufgrund einer zugeordneten IComparer-Implementierung sortiert wird.
Stack	Stellt ein Last-In-First-Out(LIFO)-Auflistungsobjekt dar.

Wichtig dabei ist auch die Betrachtung der Interfaces, welche die unterschiedlichen Generic-Klassen implementieren. Diese will ich in Tabelle 6.2 kurz darstellen.

Tabelle 6.2 Interfaces im Namespace System.Collections.Generic

Interface	Beschreibung
ICollection	Definiert folgende Methoden für generische Auflistungen: ■ Add ■ Clear ■ Contains ■ CopyTo ■ Count (schreibgeschützte Eigenschaft) ■ IsReadOnly (schreibgeschützte Eigenschaft) ■ Remove Implementiert außerdem das Interface IEnumerable.
IComparer	Definiert eine Methode `Compare` zum Vergleichen zweier Objekte.
IDictionary	Definiert folgende Methoden für generische Auflistungen mit Schlüssel-Werte-Paar-Beziehungen: ■ Add (zusätzliche Überladung) ■ ContainsKey ■ Item (Eigenschaft) ■ Keys (Eigenschaft) ■ Remove (zusätzliche Überladung) ■ TryGetValue ■ Values (schreibgeschützte Eigenschaft) Implementiert außerdem die Interfaces ICollection und IEnumerable.
IEnumerable	Gibt den Enumerator zum Durchlaufen der Collection mittels der Methode `GetEnumerator` bekannt.
IEnumerator	Stellt folgende Methoden für den Enumerator zur Verfügung (wird benötigt für die For Each-Schleife): ■ Current ■ MoveNext ■ Reset
IEqualityComparer	Stellt folgende beiden Methoden zur Verfügung, um Objekte auf Gleichheit zu überprüfen: ■ Equals ■ GetHashCode

Tabelle 6.2 (Forts.) Interfaces im Namespace System.Collections.Generic

IList	Stellt folgende Methoden zur Verfügung, um auf Objekte über Indizes zuzugreifen: ■ IndexOf ■ Insert ■ Item (Eigenschaft) ■ RemoveAt Implementiert außerdem die Interfaces IEnumerable und ICollection.

In der Tabelle 6.3 will ich noch kurz auflisten, welche Klassen aus Tabelle 6.1 welche Interfaces aus Tabelle 6.2 implementieren.

Tabelle 6.3 Übersicht der in den Generic-Klassen implementierten Interfaces

Klasse	Implementierte Interfaces
Dictionary	IDictionary, ICollection, IEnumerable
LinkedList	ICollection, IEnumerable
List	IList, ICollection, IEnumerable
Queue	IEnumerable
SortedDictionary	IDictionary, ICollection, IEnumerable
SortedList	IDictionary, ICollection, IEnumerable
Stack	IEnumerable

Somit haben Sie hoffentlich eine Vorstellung, welchen Funktionsumfang die bestimmten Generic-Klassen besitzen.

Nun will ich Ihnen noch in einer kleinen Applikation die Syntax für Generics am Beispiel einer `SortedList` zeigen.

Dazu habe ich eine Klasse `Person` geschrieben, die nur drei Eigenschaften `Vorname`, `Famname` und `Geburtsdatum` besitzt. Der Konstruktor dieser Klasse erwartet zwei Parameter vom Typ `String` für den Vor- und Familiennamen.

In meiner Applikation definiere und instanziiere ich ein Objekt `Personen` vom Typ `SortedList`.

```
Private Personen As New _
    System.Collection.Generic.SortedList(Of String, Person)
```

Wie Sie sehen, wird mittels eines `Of`-Operators dieser Klasse mitgeteilt, von welchem Datentyp der Schlüssel und von welchem Datentyp die in dieser Auflistung enthaltenen Objekte sind.

Ich kann somit nur noch Objekte vom Typ `Person`, oder von Person abgeleitete Typen, die als Schlüssel einen `String` besitzen, zu dieser Collection hinzufügen. Mehr muss ich nicht tun.

Danach füge ich der Auflistung zwei Personenobjekte hinzu. Würde ich hier einen Schlüssel angeben, der nicht vom Datentyp `String` ist, oder versuchen, ein anderes Objekt hinzuzufügen, würde ein Compiler-Fehler erzeugt werden und kein Laufzeitfehler.

Wenn ich nun über die `Items`-Eigenschaft auf ein bestimmtes Personenobjekt zugreifen will, sehen Sie in Abbildung 6.1, dass ich keine Typumwandlung durchführen muss und außerdem eine vollständige IntelliSense-Unterstützung habe.

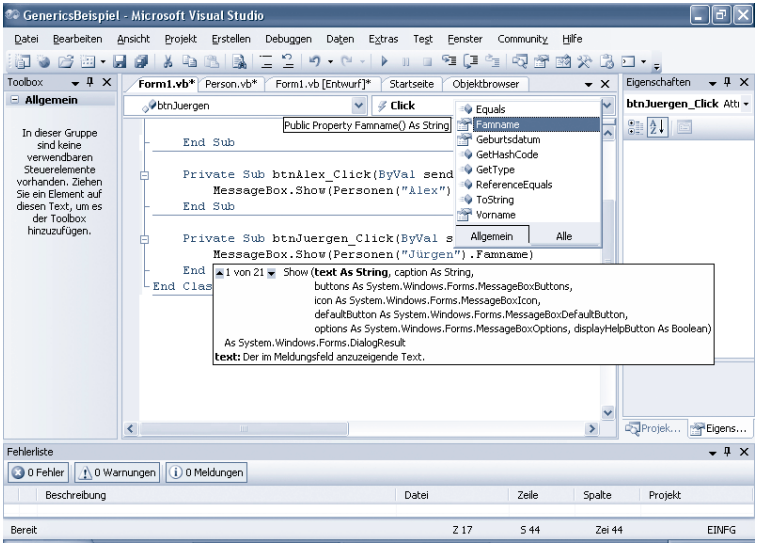


Abbildung 6.1
IntelliSense-Unterstützung bei Generics

Den kompletten Programmcode zu dem Beispiel sehen Sie in Listing 6.1.

Listing 6.1
Beispielcode für Generics

```
Public Class Form1
    Private Personen As New _
        System.Collections.Generic.SortedList _
        (Of String, Person)

    Private Sub Form1_Load(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

        Personen.Add("Alex", New Person("Alex", "Bierhaus"))
        Personen.Add("Jürgen", New Person("Jürgen", "Kotz"))
    End Sub

    Private Sub btnAlex_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles btnAlex.Click

        MessageBox.Show(Personen("Alex").Famname)
    End Sub
```

Listing 6.1 (Forts.)
Beispielcode für Generics

```
    Private Sub btnJuergen_Click(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles btnJuergen.Click

        MessageBox.Show(Personen("Jürgen").Famname)
    End Sub
End Class

Public Class Person
    Private mVorname As String
    Private mFamname As String
    Private mGeburtsdatum As Date

    Public Sub New(ByVal vorName As String, _
        ByVal famName As String)
        Me.Vorname = vorName
        Me.Famname = famName
    End Sub
    Public Property Vorname() As String
        Get
            Return mVorname
        End Get
        Set(ByVal value As String)
            mVorname = value
        End Set
    End Property
    Public Property Famname() As String
        Get
            Return mFamname
        End Get
        Set(ByVal value As String)
            mFamname = value
        End Set
    End Property
    Public Property Geburtsdatum() As Date
        Get
            Return mGeburtsdatum
        End Get
        Set(ByVal value As Date)
            mGeburtsdatum = value
        End Set
    End Property
End Class
```

Dieses Beispiel soll zeigen, wie einfach grundsätzlich Generics anzuwenden sind und warum Sie diesen Namespace auch nutzen sollten, wenn Sie mit Auflistungsklassen arbeiten.

Nicht zu unterschätzen ist wirklich der Performancegewinn gegenüber den herkömmlichen Collection-Klassen. Da der Compiler den Code bezüglich der benutzten Typen optimiert, werden keine laufzeitschädlichen Typumwandlungen mehr durchgeführt.

Hinweis

Generics verwenden auch kein **Boxing** und **Unboxing**. Unter Boxing versteht man das Umwandeln eines Wertetyps in einen Objekttyp, Unboxing ist die Rückumwandlung zurück in einen Wertetyp. Dieser Vorgang wird bei den herkömmlichen Auflistungsklassen durchgeführt, wenn einfache Typen gespeichert werden müssen, da diese zu einem `Object` und danach wieder zurück konvertiert werden müssen.

Doch die ganze Sache geht noch einen Schritt weiter. Denn Sie können nicht nur die Generic-Klassen nutzen, Sie können auch selbst eigene Typen und Methoden erstellen.

Generische Methoden sind zum Beispiel dann sinnvoll, wenn Sie mittels einer Methode, unabhängig vom Datentyp der übergebenen Parameter, einen bestimmten Algorithmus anwenden wollen.

Ein klassisches Beispiel hierfür ist ein **BubbleSort**-Algorithmus.

Bei diesem Algorithmus werden jeweils zwei Werte miteinander verglichen und bei Bedarf getauscht. Und genau die Routine, in der die Werte vertauscht werden, wollen wir als generische Methode definieren. Dies hat den Vorteil der Wiederverwendbarkeit des geschriebenen Codes für unterschiedliche Datentypen.

Innerhalb einer Konsolenanwendung generiere ich per Zufallsgenerator ein Array mit zehn Zufallszahlen vom Typ `Integer`. Mittels des Bubble-Sort-Algorithmus will ich die Werte sortieren und verwende dazu eine Methode `Wechsle`, die zwei Werte bei Bedarf miteinander tauscht. Die Methode `Wechsle` ist dabei als generische Methode definiert. Das Beispiel finden Sie in Listing 6.2.

Listing 6.2
Beispiel für eine generische Methode

```
Sub Main()
    Dim zahlen(9) As Integer
    Dim r As New Random
    'Zahlengenerierung
    For i As Integer = 0 To 9
        zahlen(i) = r.Next(0, 100)
    Next
    'Sortierung
    For i As Integer = 0 To 9
        For j As Integer = 9 To 1 Step -1
            If zahlen(j) < zahlen(j - 1) Then
                Wechsle(Of Integer)(zahlen(j), zahlen(j - 1))
            End If
        Next
    Next
    'Ausgabe
    For i As Integer = 0 To 9
        Console.WriteLine(zahlen(i))
    Next
    Console.ReadLine()
End Sub
```

Listing 6.2 (Forts.) Beispiel für eine generische Methode

```
Private Sub Wechsle(Of ItemType) _
    (ByRef item1 As ItemType, ByRef item2 As ItemType)

    Dim temp As ItemType
    temp = item1
    item1 = item2
    item2 = temp
End Sub
```

An den `Of`-Operator wird ein beliebiger Typ übergeben. Die Bezeichnung `ItemType` wird beim Aufruf an jeder Stelle innerhalb des Programmcodes durch den tatsächlich übergebenen Typ ersetzt. Der Datentyp muss dabei vom Aufrufer zusätzlich zu den beiden anderen Parametern übergeben werden, wie Sie in der `Main`-Routine sehen können.

```
Wechsle(Of Integer)(zahlen(j), zahlen(j - 1))
```

Der JIT-Compiler generiert bei diesem Aufruf folgenden Programmcode:

```
Private Sub Wechsle(ByRef item1 As Integer, _
    ByRef item2 As Integer)
    Dim temp As Integer
    temp = item1
    item1 = item2
    item2 = temp
End Sub
```

Wäre der Aufruf mit `Of String` erfolgt, so hätte der JIT-Compiler an den entsprechenden Stellen `string`-Variablen verwendet.

Genauso gut können Sie auf dieselbe Art und Weise auch eigene generische Typen definieren:

```
Public Class MyGenericClass(Of ItemType)
```

Wobei an dieser Stelle `ItemType` an jeder beliebigen Stelle innerhalb dieser Klasse als Datentyp verwendet werden kann, der dann vom Compiler durch den an den `Of`-Operator übergebenen Typ ersetzt wird.

Generics sind mein persönlicher Favorit in .NET 2.0, denn sie bieten wesentliche Verbesserungen bezüglich Laufzeitverhalten, strikter Typenüberprüfung und IntelliSense-Unterstützung gegenüber den bislang etablierten Collection-Objekten.

6.2 Das My-Object

Das `My-Object` ist eine weitere Neuerung in Visual Basic 2005, die den Entwicklern Arbeit und Komplexität abnehmen soll.

Hinweis

Im Gegensatz zu Generics steht `My` aber nur in Visual Basic 2005 zur Verfügung, für C# oder andere .NET-Sprachen wurde es nicht implementiert.

Mit My hat man einen einfachen und schnellen Zugriff auf wichtige Klassen, die ansonsten in den Tiefen der Framework-Klassenbibliothek verstreut sind.

My bietet dabei einen Zugriff auf folgende Objekte:

- Application

  My.Application bietet Zugriff auf Daten, die dieser Applikation zugeordnet sind (wie zum Beispiel Kulturinformationen, Befehlszeilenargumente, alle geöffneten Formulare).

- Computer

  My.Computer bietet Zugriff auf Computerkomponenten (wie zum Beispiel Zwischenablage, Uhr, Tastatur, Filesystem).

- Forms

  My.Forms bietet Zugriff auf ein instanziiertes Formular innerhalb einer Windows-Applikation. My.Forms ist auch nur in Windows-Anwendungen verfügbar.

- Resources

  My.Resources bietet schreibgeschützten Zugriff auf Ressourcen Ihrer Applikation (wie zum Beispiel Bilder und Zeichenfolgen für Lokalisierung).

- Settings

  My.Settings bietet lesenden und schreibenden Zugriff auf Anwendungseinstellungen.

- User

  My.User bietet Zugriff auf Informationen des aktuellen Benutzers (wie zum Beispiel Name, Windows-Gruppen).

- WebServices

  My.WebServices bietet Zugriff auf jede Webservice-Instanz, auf die aus Ihrem Projekt referenziert wird.

Im Folgenden will ich zu einigen My-Objekten ein kleines Beispiel zeigen, um einen Überblick über die Mächtigkeit dieses Konstruktes zu zeigen. Dabei werden in einer ListView verschiedene Einträge über das entsprechende Objekt in die Liste eingetragen.

Beginnen wir mit My.Application und Listing 6.3.

Listing 6.3
Beispiel für My.Application

```
Dim li As New ListViewItem
li.SubItems.Add(My.Application.CommandLineArgs.Count. _
    ToString)
li.Text = "Anzahl Befehlszeilenargumente"
ListView1.Items.Add(li)

li = New ListViewItem
li.Text = "Kultur"
li.SubItems.Add(My.Application.Culture.ToString)
ListView1.Items.Add(li)
```

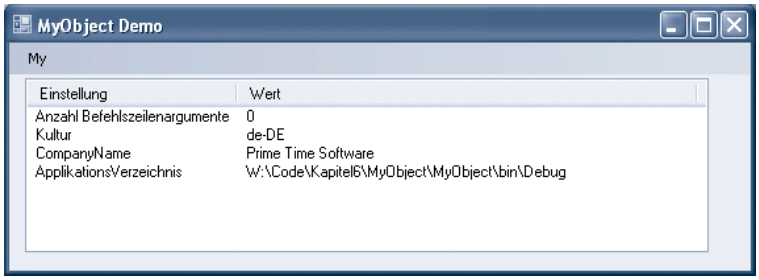
Listing 6.3 (Forts.)
Beispiel für My.Application

```
li = New ListViewItem
li.Text = "CompanyName"
li.SubItems.Add(My.Application.Info.CompanyName)
ListView1.Items.Add(li)

li = New ListViewItem
li.Text = "ApplikationsVerzeichnis"
li.SubItems.Add(My.Application.Info.DirectoryPath)
ListView1.Items.Add(li)
```

Das Ergebnis sehen Sie in Abbildung 6.2.

Abbildung 6.2
My.Application



Bei den folgenden Codebeispielen will ich nur noch den Code darstellen, der den Aufruf an das `My`-Objekt durchführt.

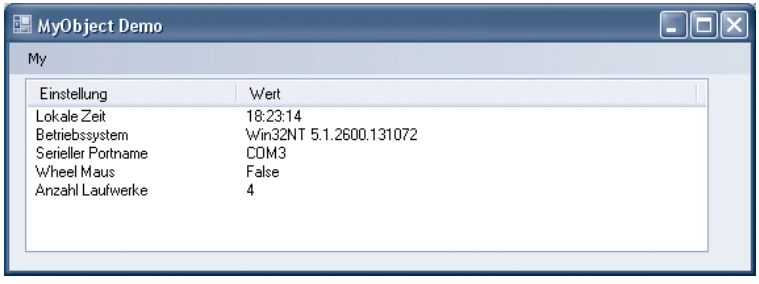
Listing 6.4 zeigt ein Beispiel für `My.Computer`.

Listing 6.4
Beispiel für My.Computer

```
li.SubItems.Add(My.Computer.Clock.LocalTime.ToLongTimeString)
li.SubItems.Add(My.Computer.Info.OSPlatform & " " & _
    My.Computer.Info.OSVersion)
li.SubItems.Add(My.Computer.Ports.SerialPortNames(0). _
    ToString)
li.SubItems.Add(My.Computer.Mouse.WheelExists.ToString)
li.SubItems.Add(My.Computer.FileSystem.Drives.Count.ToString)
```

Abbildung 6.3 zeigt die Ausgabe für `My.Computer`.

Abbildung 6.3
My.Computer



Nur in Windows-Applikationen verfügbar ist `My.Forms`.

Listing 6.5
Beispiel für My.Forms

```
li.SubItems.Add(My.Forms.Formulartest.Text)
li.SubItems.Add(My.Forms.Formulartest.IsMdiContainer. _
    ToString)
li.SubItems.Add(My.Forms.Formulartest.BackColor.G. _
```

```
    ToString)
li.SubItems.Add(My.Forms.Formulartest.Controls.Count. _
    ToString)
```

Listing 6.5 (Forts.)
Beispiel für My.Forms

Hierfür habe ich ein zweites Formular `Formulartest` zur Applikation hinzugefügt. Abbildung 6.4 zeigt die entsprechende Ausgabe.

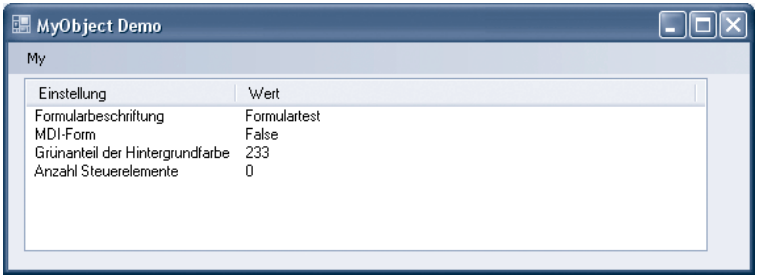


Abbildung 6.4
My.Forms

Und zu guter Letzt noch ein Beispiel für `My.User`.

```
li.SubItems.Add(My.User.Name)
li.SubItems.Add(My.User.IsInRole _
    (ApplicationServices.BuiltInRole.Administrator.ToString))
li.SubItems.Add(My.User.CurrentPrincipal. _
    Identity.AuthenticationType.ToString)
li.SubItems.Add(My.User.CurrentPrincipal. _
    Identity.IsAuthenticated)
```

Listing 6.6
Beispiel für My.User

Was zur Bildschirmausgabe in Abbildung 6.5 führt.

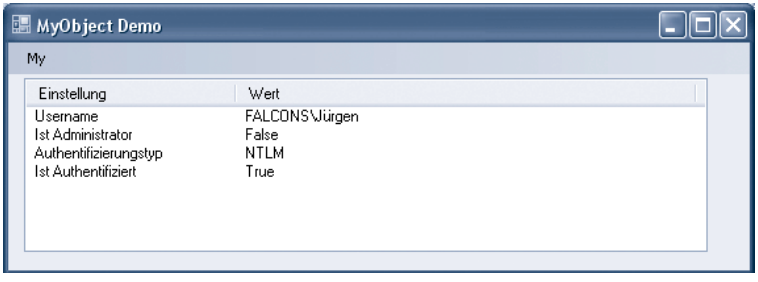


Abbildung 6.5
My.User

Zu erwähnen ist, dass es über das `My`-Objekt nicht nur möglich ist, Eigenschaften aus zentralen Stellen auszulesen oder zu setzen, sondern auch Funktionalität auszuführen.

Die Anweisung

```
My.Computer.FileSystem.CreateDirectory("C:\test")
```

würde zum Beispiel ein neues Verzeichnis *test* anlegen, wobei `My.Computer.FileSystem` einen vollständigen Zugriff (falls die Sicherheitseinstellungen es zulassen) auf das Dateisystem des Rechners gewährt.

Während man mit

```
My.Computer.Network.Ping(InputBox(InputBox("IP-Adresse"))
```

einen **Ping** auf einen anderen Rechner absetzen kann.

Die Möglichkeiten sind sehr vielfältig und erleichtern an vielen Stellen die Arbeit wirklich enorm.

6.3 Operator Overloading

Operatorenüberladung hat es ja bei C# schon in der ersten Version gegeben. Mit Visual Basic 2005 steht dieses Feature jetzt auch den Basic-Programmierern zur Verfügung.

Mittels des Überladens von Operatoren können Sie das Verhalten Ihrer Klasse definieren, wenn diese mit einem Operator aufgerufen werden. Das durch ein + zum Beispiel zwei Strings konkateniert werden, ist sicherlich jedem einleuchtend, aber was passiert, wenn Sie den +-Operator auf eigene definierte Klassen einsetzen? Bislang haben Sie einen Compiler-Fehler erhalten, aber diese Zeiten können vorbei sein, denn mittels des Überladens von Operatoren können Sie für Ihre eigenen Klassen bestimmte Funktionalität hinterlegen.

Sie können unter anderem folgende Operatoren überladen +, –, <, <=, >, >=, =, <>, *, /.

Nehmen wir an, wir haben eine Artikelklasse und wollen ein Array von Artikeln nach Preisen sortieren. Dazu wollen wir denselben Bubble-Sort-Algorithmus wie in Listing 6.2 verwenden (inklusive unserer generischen `Wechsle`-Methode).

Um das zu realisieren, müssen wir den >- und <-Operator für die Klasse `Artikel` überschreiben. Wobei das Definieren von überladenen Operatoren eigentlich ganz einfach ist.

Sie machen das, als würden Sie eine ganz herkömmliche Methode anlegen, anstatt `Sub` oder `Function` schreiben Sie jedoch `Operator` und statt den Methodennamen geben Sie den Operator an. Der Operator muss zwingend als statische Methode mit dem Schlüsselwort `Shared` gekennzeichnet sein, und einer der übergebenen Parameter muss vom Typ der entsprechenden Klasse sein. Wenn es mehrere Parameter gibt (was bei Vergleichsoperatoren wie <, >, =, >=, <= ja durchaus Sinn macht), muss zumindest einer von beiden vom entsprechenden Typ sein.

Innerhalb des überladenen Operators schreiben Sie einfach die Logik, die Sie an dieser Stelle einsetzen wollen.

Achtung

Achten Sie bitte darauf, dass Sie innerhalb der Logik die Parameter auch auf `Nothing` überprüfen, da eventuell eine Objektvariable noch nicht zugewiesen ist und Sie an dieser Stelle keinen Laufzeitfehler riskieren wollen.

Bestimmte Operatoren lassen sich nicht alleine überladen. Wenn Sie zum Beispiel den <-Operator überladen, müssen Sie auch den >-Operator überladen, was durchaus seine Konsequenz hat. Machen Sie das nicht, resultiert die ganze Sache in einem Compiler-Fehler.

Im Listing 6.7 sehen Sie die Klassendefinition für unsere Artikelklasse inklusive Überladung für die Operatoren <, >, =, <>.

Listing 6.7
Beispiel für Operator Overloading

```
Public Class Artikel
    Private mBezeichnung As String
    Private mPreis As Double
    Public Property Bezeichnung() As String
        Get
            Return mBezeichnung
        End Get
        Set(ByVal value As String)
            mBezeichnung = value
        End Set
    End Property
    Public Property Preis() As Double
        Get
            Return mPreis
        End Get
        Set(ByVal value As Double)
            mPreis = value
        End Set
    End Property

    Public Shared Operator >(ByVal Artikel1 As Artikel, _
        ByVal Artikel2 As Artikel) As Boolean

        If Artikel1 Is Nothing And Artikel2 Is Nothing Then
            Return False
        End If
        If Artikel1 Is Nothing Then
            Return False
        End If
        If Artikel2 Is Nothing Then
            Return True
        End If
        Return Artikel1.Preis > Artikel2.Preis
    End Operator

    Public Shared Operator <(ByVal Artikel1 As Artikel, _
        ByVal Artikel2 As Artikel) As Boolean

        If Artikel1 Is Nothing And Artikel2 Is Nothing Then
            Return False
```

Listing 6.7 (Forts.) Beispiel für Operator Overloading

```
            End If
            If Artikel1 Is Nothing Then
                Return True
            End If
            If Artikel2 Is Nothing Then
                Return False
            End If
            Return Artikel1.Preis < Artikel2.Preis
        End Operator

        Public Shared Operator =(ByVal Artikel1 As Artikel, _
            ByVal Artikel2 As Artikel) As Boolean

            If Artikel1 Is Nothing And Artikel2 Is Nothing Then
                Return True
            End If
            If Artikel1 Is Nothing Then
                Return False
            End If
            If Artikel2 Is Nothing Then
                Return False
            End If
            Return Artikel1.Preis = Artikel2.Preis
        End Operator

        Public Shared Operator <>(ByVal Artikel1 As Artikel, _
            ByVal Artikel2 As Artikel) As Boolean

            If Artikel1 Is Nothing And Artikel2 Is Nothing Then
                Return False
            End If
            If Artikel1 Is Nothing Then
                Return True
            End If
            If Artikel2 Is Nothing Then
                Return True
            End If
            Return Artikel1.Preis <> Artikel2.Preis
        End Operator
    End Class
```

In der `Sub Main` der Konsolenanwendung wollen wir jetzt diese Operatorenüberladung nutzen und die Artikel nach Preis sortieren. Den Programmcode sehen Sie in Listing 6.8.

Listing 6.8 Operator Overloading in der Praxis

```
Dim Produkte(9) As Artikel
Dim r As New Random
'Array befüllen
For i As Integer = 0 To 9
    Dim a As New Artikel
    a.Bezeichnung = "Artikel" & i.ToString
    a.Preis = r.NextDouble() * 100
    Produkte(i) = a
Next
'Sortierung
For i As Integer = 0 To 9
    For j As Integer = 9 To 1 Step -1
```

Listing 6.8 (Forts.)
Operator Overloading in der Praxis

```
        If Produkte(j) < Produkte(j - 1) Then
            Wechsle(Of Artikel)(Produkte(j), Produkte(j - 1))
        End If
    Next
Next
'Ausgabe
For i As Integer = 0 To 9
    Console.WriteLine(Produkte(i).Bezeichnung & "  " & _
        Produkte(i).Preis.ToString("##0.00"))
Next
Console.ReadLine()
```

Wir können jetzt die Produkte mit den von uns überladenen Operatoren bequem vergleichen. Ich finde es auch schön, wie unsere generische `Wechsle`-Methode auch für Artikel funktioniert.

Sicherlich hätten wir dieses Problem auch ohne das Überladen von Operatoren relativ leicht lösen können, doch ich wollte das Beispiel bewusst einfach halten. Und je aufwändiger die dahinter liegende Logik ist, desto mehr macht es Sinn, dieses neue Feature auch einzusetzen. Beachten Sie bitte nur, dass das Lesen des Programmcodes auch durch den Einsatz dieser Technologie noch intuitiv bleiben soll, denn Code wird bei weitem öfter gelesen als geschrieben. Also übertreiben Sie es nicht mit dem Operator Overloading.

6.3.1 Überladen von CType

Eine spezielle Art der Operatorenüberladung gibt es für den `CType`-Operator.

Sie können definieren, wie Ihr Objekt in jeden beliebigen einfachen Datentyp gewandelt wird, indem Sie den `CType`-Operator erweitern.

Ich habe für unsere Artikelklasse eine Konvertierung auf `String` und auf `Double` erweitert, so dass, wenn ich einen `Artikel` in einen `String` umwandle, die Artikelbezeichnung zurück übergeben wird und bei einer Umwandlung nach `Double` der entsprechende Preis des Artikels.

Listing 6.9 zeigt die Erweiterung der Artikelklasse.

Listing 6.9
Erweiterung der CType-Überladung

```
Public Shared Widening Operator CType _
    (ByVal art As Artikel) As String

    If art Is Nothing Then
        Return ""
    Else
        Return art.Bezeichnung
    End If
End Operator
Public Shared Widening Operator CType _
    (ByVal art As Artikel) As Double
    If art Is Nothing Then
        Return 0.0
    Else
        Return art.Preis
    End If
End Operator
```

Mit dem Schlüsselwort `Widening` wird die Überladung für den `CType`-Operator erweitert.

Sie können jetzt eine Objektinstanz von Artikeln mit folgender Anweisung umwandeln:

```
Dim x As String = CType(artikel, String)
```

oder

```
Dim y As Double = CType(artikel, Double)
```

6.3.2 IsTrue- und IsFalse-Operatoren

Interessant ist an dieser Stelle auch noch der Hinweis auf die Überladungsmöglichkeit für zwei Operatoren, die Sie im Code direkt gar nicht benutzen dürfen und die Ihnen deswegen wohl auch nicht bekannt sind.

`IsTrue` und `IsFalse` sind zwei Operatoren, die nur für Überladung zur Verfügung stehen.

Mittels dieser Überladungen ist es für den Compiler möglich, einen booleschen Wert zu bestimmen. Dies geschieht in dem Fall, dass ein Objekt an einer Stelle benutzt wird, an der ein boolescher Wert erwartet wird, wie zum Beispiel in folgender Abfrage `If art Then`.

Bislang würde der Compiler eine Fehlermeldung ausgeben, dass der Typ nicht zu `Boolean` konvertiert werden kann. Wenn Sie jedoch die `IsTrue`- und `IsFalse`-Operatoren überladen, wird der Compiler genau dies können.

Listing 6.10 zeigt, wie Sie diese Überladung im Code implementieren können.

Listing 6.10 Überladung von IsTrue und IsFalse

```
Public Shared Operator IsTrue _
    (ByVal art As Artikel) As Boolean

    If art Is Nothing Then
        Return False
    Else
        Return True
    End If
End Operator
Public Shared Operator IsFalse _
    (ByVal art As Artikel) As Boolean

    If art Is Nothing Then
        Return True
    Else
        Return False
    End If
End Operator
```

Das bedeutet, wenn jetzt die Objektvariable auf einen booleschen Wert abgefragt wird, ist der Rückgabewert in Abhängigkeit, ob die Variable bereits einer Instanz zugewiesen ist, `true` oder `false`. Dank dieser Überla-

dung brauchen Sie eigentlich jetzt nicht mehr auf `Is Nothing` abzuprüfen, das erledigt diese Überladung für Sie. Und wenn wir jetzt schon dabei sind, überladen wir auch gleich noch den `Not`-Operator, wie Sie in Listing 6.11 sehen.

Listing 6.11 Überladung des Not-Operators

```
Public Shared Operator Not(ByVal art As Artikel) As Boolean
    If art Then
        Return False
    Else
        Return True
    End If
End Operator
```

So, das wär's gewesen zu Operator Overloading, und jetzt schauen wir uns noch weitere neue Sprachmerkmale in Visual Basic 2005 an.

6.4 Sonstige neue Sprachelemente

Zum Abschluss dieses Kapitels will ich Ihnen noch einen Überblick über die wichtigsten neuen Sprachelemente und Schlüsselwörter geben, die in Visual Basic 2005 eingeführt wurden.

6.4.1 IsNot-Operator

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

```
If obj IsNot Nothing Then
```

als

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

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

6.4.2 Neue Schlüsselwörter

In Visual Basic 2005 wurden drei neue Schlüsselwörter eingeführt, die wir uns hier noch kurz betrachten sollten.

Global

Bevor ich Ihnen das Schlüsselwort `Global` näher erläutere, erlauben Sie mir vorab eine kleine Bemerkung. Ich hoffe nur, dass Sie dieses Schlüsselwort nie benötigen.

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

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

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

Wenn Sie in Ihrem eigenen Namespace nun folgende Variablendefinition vornehmen:

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

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

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

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

Using

Da wir in .NET ein nichtdeterministisches Verhalten des Garbage Collectors haben, können wir nicht vorhersehen, zu welchem Zeitpunkt unsere Objekte wieder im Speicher freigegeben werden. Der Zeitpunkt liegt zwar in der nahen Zukunft, aber unsere Objekte halten solange noch Zugriff auf Ressourcen, die diese beansprucht haben. Um diese Ressourcen bereits vorzeitig freizugeben, können Sie für Ihre Objekte das `IDisposable`-Interface implementieren und eine Methode `Dispose` bereitstellen (siehe Kapitel 0), aber Sie können nicht beeinflussen, dass der Entwickler, der Ihre Objekte nutzt, die Methode `Dispose` auch aufruft.

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

Das Schlüsselwort `Using` verwenden Sie dabei wie in Listing 6.12.

Listing 6.12 Verwendung von Using

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

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

Sie können aber auch alternativ bei der Verwendung von `Using` das entsprechende Objekt erst instanziieren:

```
Using a As New Artikel
```

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

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

Continue

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

Achtung

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

Wenn Sie zum Beispiel Datensätze aus einer Datei oder einem `DataSet` durchlaufen und in Abhängigkeit von einem Wert soll eine Verarbeitung nicht stattfinden, können Sie hier mit `Continue` den verarbeitenden Block überspringen. Es wird dann ans Schleifenende gesprungen und der nächste Datensatz verarbeitet. Listing 6.13 zeigt eine Verwendung von `Continue`.

Listing 6.13
Verwendung von Continue

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

`Continue` können Sie dabei für jeglichen Schleifentyp verwenden.