

Kapitel 6

Generics (Generika) und generische Auflistungen

In diesem Kapitel:

Einführung	110
Lösungsansätze	111
Typengeneralisierung durch den Einsatz generischer Datentypen	113
Beschränkungen (Constraints)	116
Generische Auflistungen (Generic Collections)	126
List(Of)-Auflistungen und Lambda-Ausdrücke	128

Einführung

Generics – oder »Generika« wie Microsoft die so schöne deutsche Übersetzung dafür fand – sind nicht neu in Visual Basic 2008. Das erste Argument, sie dennoch in diesem Buch zu behandeln, ist, dass Generics und vor allem generische Auflistungen entscheidend für das Verständnis des nächsten Kapitels sind, in dem es um die Grundlagen zu LINQ geht. Das zweite Argument: Nicht jeder VB-Entwickler hat bereits mit Generics und generischen Auflistungen gearbeitet – eine Veranschaulichung der Konzepte kann daher so oder so nicht schaden, denn auch wenn Sie nicht vorhaben, in Zukunft mit LINQ zu arbeiten: das Verstehen von Generics und das Verwenden generischer Auflistungen wird Ihnen in jedem Fall helfen, Ihre Anwendungen sicherer zu machen und sie damit auch professionellen Standards genügen zu lassen.

Generics: Verwenden einer Codebasis für verschiedene Typen

Wenn Sie Methoden und Eigenschaften entwickeln, haben diese unter Umständen einen Nachteil: Sie verarbeiten, wenn sie typischer sein sollen, nur einen bestimmten Datentyp.

Die Alternative dazu ist, dass Sie einen Typ schaffen, der die Aufnahme beliebiger Datentypen durch den auf Object basierenden Einsatz ermöglicht, doch ein solcher Typ ist dann nicht typischer und kann zur Laufzeit Ausnahmen auslösen, mit denen Sie nicht rechnen.

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis `.\Samples\Chapter06 - Generics\Generics01`.

Dieses Beispiel verwendet die aus dem vorherigen Kapitel bekannte Klasse `DynamicList` in leicht modifizierter Form. Das Modul verwendet eine Instanz von `DynamicList` und fügt ihr ein paar Elemente hinzu. Diese Elemente gibt es anschließend in einer `For/Each`-Schleife wieder im Konsolenfenster aus:

```
Module mdlMain

    Sub Main()
        Dim locListe As New DynamicList
        locListe.Add(123.32)
        locListe.Add(126.32)
        locListe.Add(124.52)
        locListe.Add(29.99)
        locListe.Add(13.54)
        'Der wird Probleme machen!
        locListe.Add(#12/31/2005 4:00:00 PM#)
        locListe.Add(43.32)
        For Each locItem As Double In locListe
            Console.WriteLine(locItem)
        Next

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub

End Module
```

Wenn Sie dieses Programm starten, sehen Sie anschließend Folgendes auf dem Bildschirm:

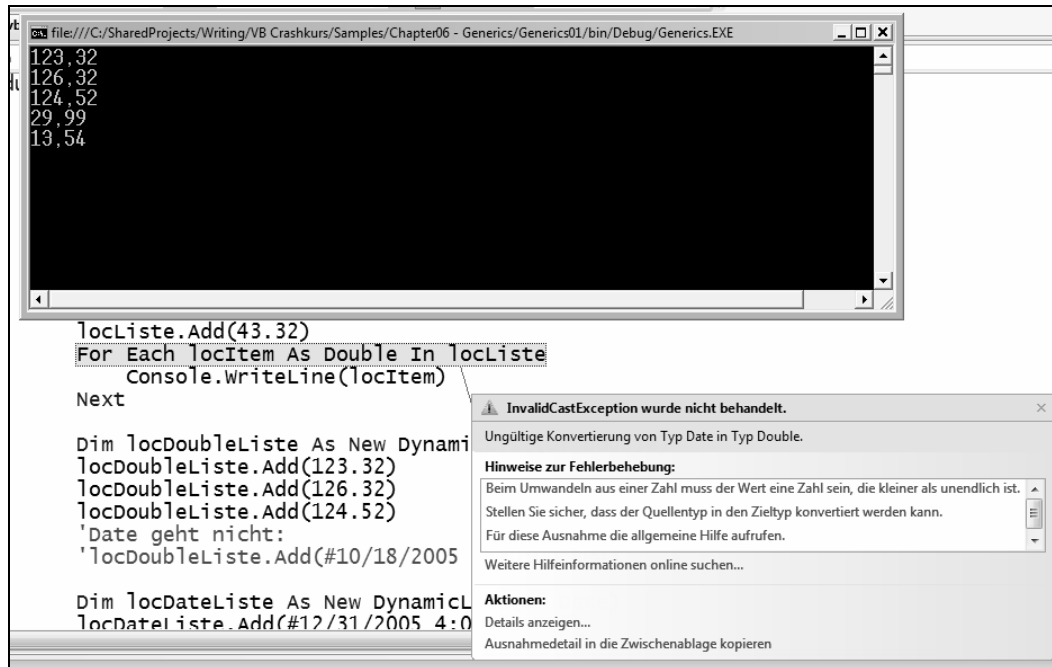


Abbildung 6.1 Die Liste wird nur bis zum *Date* Element ausgegeben; das *Date*-Element ist nämlich nicht in *Double* konvertierbar

Es ist klar, wieso das passiert. Wir haben eine Liste mit reinen *Double*-Elementen aufbauen wollen, aber uns ist ein *Date*-Element »dazwischengerutscht«. Solche Fehler sind in einem so einfachen Programm, wie wir es hier als Beispiel verwenden, noch auf den ersten Blick zu erkennen – doch in größeren Projekten sollte man solche Fehler von vornherein zu vermeiden versuchen.

Lösungsansätze

Und wie kann man das machen? Indem man eine Klasse wie die *DynamicList* typsicher macht. Indem man sie von vornherein erst gar keinen allgemein gültigen Datentyp wie *Object* akzeptieren lässt, sondern ausschließlich Werte vom Typ *Double*.

Und um das zu erreichen, nehmen wir uns den Quellcode der *DynamicList* vor, und führen die entsprechenden Änderungen durch. Konsequenterweise nennen wir diese Klasse dann auch *DynamicListDouble* (wie sie im angesprochenen Beispielprojekt übrigens bereits in einer eigenen Klassendatei vorhanden ist). Im folgenden Listing finden Sie all die Stellen in Fettschrift markiert, an denen der Datentyp *Object* in *Double* geändert wurde.

```
Class DynamicListDouble
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
```

```

Protected myCurrentCounter As Integer ' Zeiger auf aktuelles Element
Protected myArray() As Double        ' Array mit den Elementen.

Sub New()
    myCurrentArraySize = myStep
    ReDim myArray(myCurrentArraySize - 1)
End Sub

Sub Add(ByVal Item As Double)

    'Element im Array speichern
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen
    myCurrentCounter += 1

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStep

        'temporäres Array erstellen
        Dim locTempArray(myCurrentArraySize - 1) As Double
        Array.Copy(myArray, locTempArray, myArray.Length)
        myArray = locTempArray
    End If
End Sub
.
.
.
End Class

```

Wenn Sie die Klasse auf diese Weise abgeändert und das eigentliche Programm wie folgt modifiziert haben,

```

Module mdlMain

    Sub Main()
        Dim locListe As New DynamicListDouble
        locListe.Add(123.32)
        locListe.Add(126.32)
    .
    .
    .

```

zeigt Ihnen Visual Basic schon zur Entwurfszeit eine Fehlermeldung, wie Sie sie in Abbildung 6.2 sehen können.

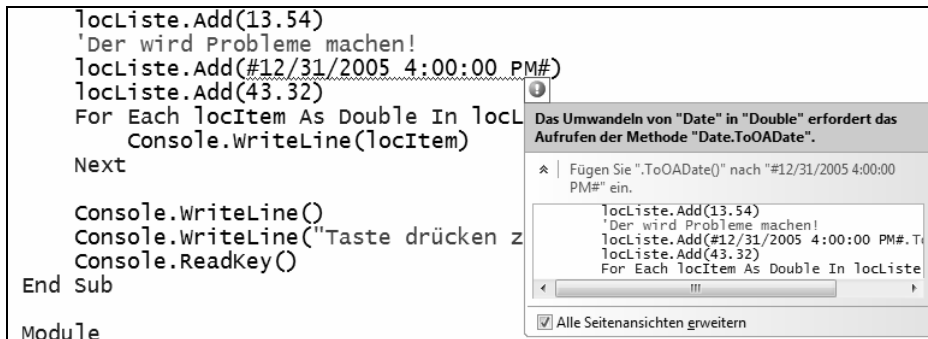


Abbildung 6.2 Bei typsicheren Klassen sehen Sie Fehlermeldungen bei »falschen« Typen bereits im Editor zur Entwurfszeit – wenngleich die Fehlermeldungen ab und an über das Ziel hinausschießen

Gut – diese Fehlermeldung an dieser Stelle schießt ein wenig über das Ziel hinaus; hätte es der Editor dabei belassen, uns über den falschen Typ an dieser Stelle zu informieren, wäre das ausreichend gewesen. Aber immerhin: Sie sehen auf jeden Fall durch die Verwendung der typsicheren Klasse schon zur Entwurfszeit, dass Sie einen Typen verwendet haben, den Sie mit dieser Klasse nicht verwenden dürfen.

Typengeneralisierung durch den Einsatz generischer Datentypen

Nun ist das Entwickeln einer Klasse wie `DynamicList` schon ein wenig aufwändiger. Und es wird ja schließlich Klassen und Strukturen geben, die noch viel, viel komplexer sind.

Doch gleichzeitig wird es gerade bei Klassen oder Strukturen, die große Datenmengen verwalten, immer wieder vorkommen, dass Sie sie für den Einsatz unterschiedlichster Typen verwenden wollen – für unser `DynamicList`-Beispiel trifft diese Aussage mehr als zu, denn:

Auch Zeichenketten ließen sich mit der Klasse wunderbar verwalten. Und auch Integer-Werte. Und auch `Decimals`. Und auch ... – eigentlich ist dieser Typ für alle Datentypen und Objektinstanzen geeignet, die Sie in größeren Mengen in einer Liste speichern wollen.

Doch als Programmierer sollten Sie aus den genannten Gründen immer auch auf Typsicherheit bestehen, und so bliebe Ihnen bislang eigentlich nur die Möglichkeit, ...

- ... für jeden benötigten Datentyp eine neue Version der verarbeitenden Klasse zu erstellen. Dieser Aufwand ist allerdings enorm groß, und zieht ein mindestens ebenso großes Problem nach sich: Finden Sie einen Fehler in einer Klasse, müssen Sie diesen in allen Klassen ändern, die sich nur durch ihren verarbeitenden Typ unterscheiden (`DynamicListDouble`, `DynamicListString`, `DynamicListDate` und welche Klasse Sie auch immer sonst noch eingerichtet hätten).
- ... eine Klasse auf Basis einer Schnittstelle zu erstellen, mit der die Typen durch Vererbung anpassbar sind. Damit hält sich der Pflegeaufwand in Grenzen, da eine Fehlerbehebung in der Basisklasse sich natürlich auch in den Klassenableitungen widerspiegelt. Doch solche Klassen zu implementieren erfordert extrem abstraktes Denken und sorgfältige Planung, und dafür steht nicht unbedingt immer die Zeit zur Verfügung.

Mit generischen Datentypen wird das anders. Bei generischen Datentypen (auf englisch »Generics« – für den Fall, dass Sie mal nach englischen Artikeln googlen müssen) legen Sie sich während der Entwicklung überhaupt noch nicht fest, welchen Typ Ihre Klasse oder Struktur später einmal verarbeiten soll. Sie arbeiten stattdessen mit so genannten Typparametern, durch die der Typ – dem JITter sei Dank – erst bei der ersten Verwendung zur Laufzeit ersetzt wird.

Mit anderen Worten: Das, was Sie mit dem Kopieren und Typanpassen Ihres Codes zur Entwicklungszeit manuell machen, dafür sorgen JITter und die Technik der Generics zur Laufzeit automatisch. Vereinfacht gesagt: So, wie Sie bei Word mit Suchen und Ersetzen arbeiten können, wird der IML-Code Ihrer generischen Klasse kopiert, und alle Typparameter werden durch den angegebenen Typ ersetzt.¹

In der Praxis und für unser `DynamicList`-Beispiel sieht das wie folgt aus:

Anstelle sich von vornherein für einen Datentyp wie `Double`, `Integer` oder `String` zu entscheiden, platzieren Sie an den entscheidenden Stellen eine Art Platzhalter – einen Typparameter –, den Sie im Kopf der Klasse mit dem Zusatz `Of` benennen, etwa so:

```
Class DynamicList(Of flexiblerDatentyp)
```

Und anstatt anschließend innerhalb der Klasse einen fixen Datentyp zu verwenden, setzen Sie diesen Typparameter als Stellvertreter ein. Für unsere `DynamicList`-Klasse bedeutet das:

```
Class DynamicList(Of flexiblerDatentyp)
    Implements IEnumerable

    Protected myStep As Integer = 4           ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexiblerDatentyp)

        myArray(myCurrentCounter) = Item
        myCurrentCounter += 1
        If myCurrentCounter = myCurrentArraySize - 1 Then
            myCurrentArraySize += myStep

            'temporäres Array erstellen
            Dim locTempArray(myCurrentArraySize - 1) As flexiblerDatentyp

            'Elemente kopieren;
            Array.Copy(myArray, locTempArray, myArray.Length)
            myArray = locTempArray
        End If
    End Sub

    'Liefert die Anzahl der vorhandenen Elemente zurück
    Public Overridable ReadOnly Property Count() As Integer
```

¹ Ganz so einfach geht es natürlich nicht. Es gibt für JITter bzw. Compiler durchaus die Möglichkeit, Codegemeinsamkeiten in generischen Klassen bestehen zu lassen, und nur dort neuen Code zu generieren, wo es nicht anders möglich ist.

```

    Get
        Return myCurrentCounter
    End Get
End Property

'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As flexiblerDatentyp
    Get
        Return myArray(Index)
    End Get
    Set(ByVal Value As flexiblerDatentyp)
        myArray(Index) = Value
    End Set
End Property

Public Function GetEnumerator() As System.Collections.IEnumerator
    Implements System.Collections.IEnumerable.GetEnumerator
    Dim locTempArray(myCurrentArraySize) As flexiblerDatentyp
    Array.Copy(myArray, locTempArray, myArray.Length)
    Return myArray.GetEnumerator
End Function
End Class

```

Und nun können Sie DynamicList für jeden Datentyp verwenden, den Sie möchten, und Sie müssen dabei nicht auf Typsicherheit verzichten, wie Abbildung 6.3 zeigt.

In der Grafik sehen Sie zweierlei. Zum einen, wie Sie einen generischen Datentyp anwenden. In der Sub Main des Moduls wird die generische Klasse einmal auf Basis des Datentyps Double definiert

```
Dim locDoubleListe As New DynamicList(Of Double)
```

und einmal auf Basis des Datentyps Date:

```
Dim locDateListe As New DynamicList(Of Date)
```

```

Dim locDoubleListe As New DynamicList(Of Double)
locDoubleListe.Add(123.32)
locDoubleListe.Add(126.32)
locDoubleListe.Add(124.52)
'Date geht nicht:
locDoubleListe.Add(#10/18/2005 3:20:00 PM#)

Dim locDateListe As New DynamicList(Of Date)
locDateListe.Add(#12/31/2005 4:00:00 PM#)
locDateListe.Add(#11/24/2005 6:20:00 PM#)
locDateListe.Add(#10/18/2005 3:20:00 PM#)
'Double geht nicht:
locDateListe.Add(124.52)

```

End
End Modu

Fehlerliste					
2 Fehler 0 Warnungen 0 Meldungen					
	Beschreibung	Datei	Zeile	Spalte	Projekt
1	Das Umwandeln von "Date" in "Double" erfordert das Aufrufen der Methode "Date.ToOADate".	mdlMain.vb	22	28	Generics
2	Das Umwandeln von "Double" in "Date" erfordert das Aufrufen der Methode "Date.FromOADate".	mdlMain.vb	29	26	Generics

Fehlerliste Codemetrikergebnisse Ausgabe

Abbildung 6.3 Mit Of bestimmen Sie, für welchen Datentyp Ihre generische Klasse zur Anwendung kommen soll

Und anhand der Fehlerliste, die Sie ebenfalls in der Grafik sehen können, erkennen Sie auch, dass beide »Typversionen« der Klasse auf ihre Weise typsicher sind. Sie können der einen nur Werte vom Typ `Double` und der anderen nur Werte vom Typ `Date` hinzufügen. Jeder Versuch, einer Liste einen jeweils anderen Typ unterzujubeln, wird schon zur Entwurfszeit mit einer entsprechenden Fehlermeldung bestraft.

HINWEIS Es gibt bei der Verwendung von Generics die Möglichkeit, den Compiler den einzusetzenden Typ auf Basis der tatsächlich übergebenden Parameter bestimmen zu lassen. Dieses Verfahren nennt man *Typrückschluss für Typparameter*, und viele Beispiele dafür finden sich bei der Verwendung der verschiedenen Überladungsversionen der Erweiterungsmethoden, die Sie in der `Enumerable`-Klasse finden, und auf denen die gesamte *Linq to Objects*-Infrastruktur basiert. Beispiele für *Typrückschluss für Typparameter* finden Sie im nächsten Kapitel.

Beschränkungen (Constraints)

Im gezeigten Beispiel können Sie eine `DynamicList` so definieren, dass sie sich aus jedem beliebigen Typ bilden kann. Unter bestimmten Umständen kann das nicht erwünscht sein, und zwar genau dann, wenn Sie innerhalb einer generischen Klasse andere generische Typen verwenden, deren Typ Sie aber zur Entwicklungszeit noch nicht kennen.

BEGLEITDATEIEN Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis `.\Samples\Chapter06 - Generics\Generics02`.

Beschränkungen für generische Typen auf eine bestimmte Basisklasse

Dazu ein Beispiel: Angenommen, Sie haben eine Anwendung geschaffen, die die verschiedensten Körper (Quader, Kugeln, Pyramiden) berechnen, verwalten und darstellen muss. Sie möchten jetzt eine generische Klasse schaffen, die nicht nur die verschiedenen Typen von Körpern in einer Liste wie der `DynamicList` speichert, sondern Sie möchten, dass diese Klasse auch deren Gesamtvolumen berechnen soll.

Nehmen wir weiter an, dass es in unserem Beispiel eine Basisklasse gibt, auf der alle Körperklassen basieren. Diese Basisklasse speichert dann die Position und die Farbe eines Körpers. Die einzelnen Körperklassen leiten anschließend von dieser Körperbasisklasse ab, damit sie die für alle gleich bleibenden Eigenschaften nicht ständig wieder implementieren müssen. Eine solche Klassenerbfolge sieht dann in etwa folgendermaßen aus:

```
Imports System.Drawing

'Stellt die Grundeigenschaften eines Körpers bereit
Public MustInherit Class KörperBasis

    Private myFarbe As Color
    Private myPosition As Point

    MustOverride ReadOnly Property Volumen() As Double

    Public Property Farbe() As Color
        Get
            Return myFarbe
        End Get
    End Property
End Class
```

```
        Set(ByVal value As Color)
            myFarbe = value
        End Set
    End Property

    Public Property Position() As Point
        Get
            Return myPosition
        End Get
        Set(ByVal value As Point)
            myPosition = value
        End Set
    End Property
End Class

'Stellt die Grundeigenschaften eines Quaders bereit
Public Class Quader
    Inherits KörperBasis

    Private mySeitenLänge_a As Double
    Private mySeitenLänge_b As Double
    Private mySeitenLänge_c As Double

    Sub New(ByVal a As Double, ByVal b As Double, ByVal c As Double)
        mySeitenLänge_a = a
        mySeitenLänge_b = b
        mySeitenLänge_c = c
    End Sub

    Public Overrides ReadOnly Property Volumen() As Double
        Get
            Return mySeitenLänge_a * mySeitenLänge_b * mySeitenLänge_c
        End Get
    End Property
End Class

'Stellt die Grundeigenschaften einer Pyramide bereit
Public Class Pyramide
    Inherits KörperBasis

    Private myGrundfläche As Double
    Private myHöhe As Double

    Sub New(ByVal Grundfläche As Double, ByVal Höhe As Double)
        myGrundfläche = Grundfläche
        myHöhe = Höhe
    End Sub

    Public Overrides ReadOnly Property Volumen() As Double
        Get
            Return (myGrundfläche * myHöhe) / 3
        End Get
    End Property
End Class
```

Rein theoretisch könnten wir natürlich nun die bereits vorhandene `DynamicList`-Klasse für die Speicherung der Körper-Objekte verwenden, wie der Code der folgenden Abbildung zeigt:

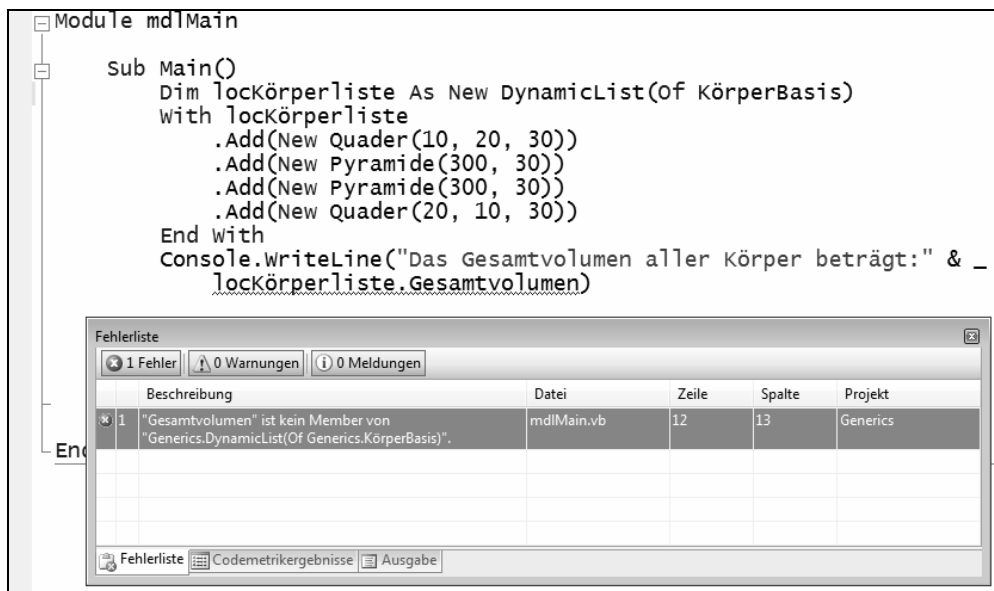


Abbildung 6.4 Die `DynamicList` soll auch eine `Gesamtvolumen`-Eigenschaft zur Verfügung stellen, doch die ist zurzeit noch nicht implementiert

Doch dabei ergibt sich eine Kette von Problemen. Wie in Abbildung 6.4 zu sehen, möchten wir eine Eigenschaft der Liste verwenden, die es zu diesem Zeitpunkt noch nicht gibt. Und mit herkömmlichen Mitteln haben wir auch leider keine Chance, diese Eigenschaft zu implementieren, denn:

Innerhalb der generischen `DynamicList`-Klasse müssten wir eine Eigenschaft `Gesamtvolumen` erschaffen, die durch alle Elemente iteriert, die sie hält, und deren `Volumen`-Eigenschaft abfragt. Doch genau eine solche Prozedur können wir nicht implementieren, wie die folgende Grafik zeigt:

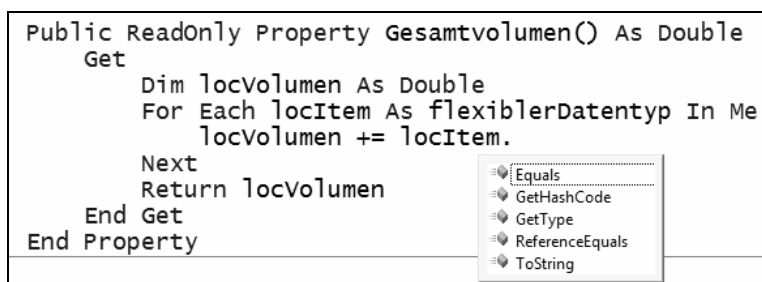


Abbildung 6.5 Die Eigenschaft, die zur Berechnung des Gesamtvolumens benötigt wird, ist nicht erreichbar!

Wenn wir die Schleife zum Iterieren durch die Elemente der `DynamicList` erstellen, dann müssen wir natürlich darauf achten, dass ein einzelnes Element dieser Iteration vom gleichen Typ ist, wie für die gesamte generische Klasse – und damit muss es vom Typ `flexiblerDatentyp` sein (anderenfalls wäre die Klasse nicht mehr generisch, also allgemeingültig anwendbar).

flexiblerDatentyp kann aber jeder beliebige Datentyp sein, und deswegen sind auch nur die Eigenschaften und Methoden erreichbar, die von jedem Datentyp *gleichermaßen* zur Verfügung gestellt werden. Das sind logischerweise genau die Eigenschaften und Methoden, die Object zur Verfügung stellt und die jede neue Klasse implizit erbt. Das wiederum sind die Elemente, die Ihnen IntelliSense, wie in Abbildung 6.5 zu sehen ist, anzeigt.

Das ist nun der richtige Zeitpunkt, Beschränkungen ins Spiel zu bringen. Wenn wir dem generischen Datentyp »sagen«, »pass auf, du darfst nur solche Datentypen als Typparameter akzeptieren, die auf Körperbasis basieren«, dann kann das .NET Framework ohne Probleme die Methoden und Eigenschaften über locItem anbieten, der vom Typ flexiblerDatentyp ist, die durch Körperbasis implementiert werden.

Diese Änderungen nehmen wir als Nächstes vor, allerdings nicht in der Klasse DynamicList selbst, denn: Damit wir nun nicht unsere DynamicList für alle Zeiten nur noch auf die Verwaltung von Körper-Objekten limitieren, implementieren wir diese Änderungen in einer Klasse, die identisch zur DynamicList-Klasse ist, jedoch nur die benötigten Änderungen noch zusätzlich innehat. Diese Klasse nennen wir DynamicListKörper, und die sieht folgendermaßen aus:

```

Class DynamicListKörper(Of flexiblerDatentyp As KörperBasis)
    Implements IEnumerable

    Protected myStep As Integer = 4           ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer  ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer    ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub

    Sub Add(ByVal Item As flexiblerDatentyp)
        .
        .
        .
    End Sub

    Public ReadOnly Property Gesamtvolumen() As Double
        Get
            Dim locVolumen As Double
            For Each locItem As flexiblerDatentyp In Me
                locVolumen += locItem.Volumen
            Next
            Return locVolumen
        End Get
    End Property
    .
    .
    .

```

Aus Platzgründen sehen Sie in diesem Listing lediglich die Änderungen im Vergleich zur Klasse DynamicList. Sie können in der ersten Zeile des Klassenlistings sehen, wie Beschränkungen implementiert werden: In den Klammern steht jetzt neben der Erweiterung Of flexiblerDatentyp, die den Typparameter für die weitere

Verwendung des generischen Typs in der Klasse festlegt, obendrein der Zusatz `as KörperBasis`, der nun bestimmt, dass ausschließlich Klassen, die von `KörperBasis` abgeleitet wurden, als Basis für die Erstellung des Datentyps `DynamicListKörper` dienen dürfen.

Das befähigt uns jetzt auch, die Eigenschaft `Gesamtvolumen` zu implementieren. Da wir die Datentypen für die generische Klasse auf `KörperBasis` und deren Ableitungen beschränken, weiß das .NET Framework, dass es alle Methoden, die `KörperBasis` anbietet, sicher für alle Objekte zur Verfügung stellen kann, die auf `flexiblerDatentyp` basieren.

Nach der Implementierung dieser neuen Klasse, stellen wir das Testprogramm im Modul `mdlMain.vb` entsprechend um:

```
Module mdlMain
    Sub Main()
        Dim lockKörperliste As New DynamicListKörper(Of KörperBasis)
        With lockKörperliste
            .Add(New Quader(10, 20, 30))
            .Add(New Pyramide(300, 30))
            .Add(New Pyramide(300, 30))
            .Add(New Quader(20, 10, 30))
        End With
        Console.WriteLine("Das Gesamtvolumen aller Körper beträgt:" & _
            lockKörperliste.Gesamtvolumen)

        Console.WriteLine()
        Console.WriteLine("Taste drücken zum Beenden!")
        Console.ReadKey()
    End Sub
End Module
```

Übrigens: Dass sich die Klasse wirklich nur noch verwenden lässt, wenn Sie sie tatsächlich auf einer Ableitung von `KörperBasis` basieren lassen, zeigt die folgende Abbildung.

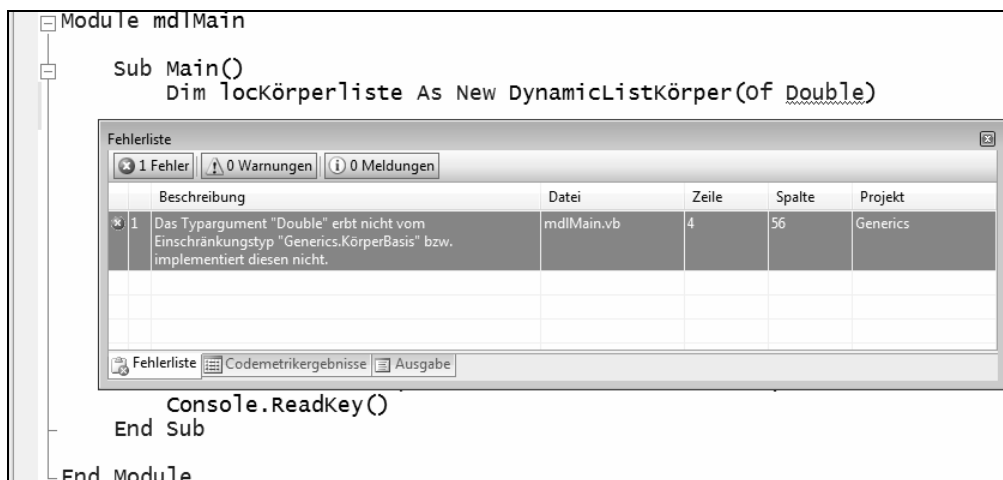


Abbildung 6.6 Eine beschränkte generische Klasse können Sie tatsächlich nur noch auf Basis eines Datentyps instanzieren, der die Beschränkung erfüllt

HINWEIS Das .NET Framework erlaubt es nicht, mehrere Basisklassen als Beschränkung für einen generischen Typ zu definieren. Das würde die Technik der Mehrfachvererbung implizieren, die das .NET Framework in der vorliegenden 2.0-Version nicht beherrscht (und wahrscheinlich auch nie beherrschen wird).

Beschränkungen auf Klassen, die bestimmte Schnittstellen implementieren

Ungleich flexibler können Sie generische Klassen gestalten, wenn sich deren generische Typen nicht auf bestimmte Basisklassen sondern nur auf bestimmte Schnittstellen beschränken.

Darüber hinaus haben Schnittstellen den Vorteil, sich bereits in vielen »fertigen« Typen des .NET Frameworks »zu befinden«, sodass Sie generische Klassen erstellen können, die nicht nur auf Ihren eigenen Typen basieren (deren Einschränkungen Sie ja gut steuern können, da Sie über deren Quellcode verfügen); vielmehr können Sie auch die Typen des .NET Frameworks verwenden, die sich, da sie die unterschiedlichsten Schnittstellen implementieren, ebenfalls sehr gut selektieren lassen.

Ein Beispiel: Sie möchten die `DynamicList`-Klasse um eine Sortierfunktion erweitern. Zu diesem Zweck müssen Sie die Elemente, die die `DynamicList`-Klasse speichert, miteinander vergleichen können. Das .NET Framework stellt für Typen, deren einzelne Instanzen sich miteinander vergleichen lassen sollen, die `IComparable`-Schnittstelle bereit. Wenn ein Typ diese Schnittstelle einbindet, zwingt ihn diese Schnittstelle damit auch, eine Funktion namens `CompareTo` einzubinden. Und wenn Sie eine generische Klasse schaffen, dann können Sie die Typen, aus denen diese hervorgehen soll, auch auf die `IComparable`-Schnittstelle beschränken. Damit bleibt die Klasse generisch, und kann dennoch jeden beliebigen Datentyp typsicher speichern – jedenfalls solange er die `IComparable`-Schnittstelle selbst implementiert. Wenn er das allerdings macht, können Sie auch vom Vorhandensein einer `CompareTo`-Funktion sicher ausgehen und damit beispielsweise eine Sortierfunktion in der generischen Klasse implementieren. Die `IComparable`-Schnittstelle wird übrigens von allen primitiven Datentypen wie `String`, `Long`, `Decimal`, `Date` etc. eingebunden.

BEGLEITDATEIEN Ein Beispiel, das Sie im Verzeichnis `.\Samples\Chapter06 - Generics\Generics03` finden, soll das verdeutlichen.

Schauen wir uns die veränderte (und aus Platzgründen wieder gekürzte) Version der `DynamicList` an, die nun den Namen `DynamicListSortable` trägt, und die – neben der Einschränkung bei der Definition des Klassennamens – um eine Sort-Methode ergänzt wurde:

```
Class DynamicListSortable(Of flexiblerDatentyp As IComparable)
    Implements IEnumerable

    Protected myStep As Integer = 4           ' Schrittweite, um die das Array erhöht wird.
    Protected myCurrentArraySize As Integer ' Aktuelle Array-Größe
    Protected myCurrentCounter As Integer   ' Zeiger auf aktuelles Element
    Protected myArray() As flexiblerDatentyp ' Array mit den Elementen.

    Sub New()
        myCurrentArraySize = myStep
        ReDim myArray(myCurrentArraySize - 1)
    End Sub
```

```

Sub Add(ByVal Item As flexiblerDatentyp)
    .
    .
    .
End Sub

'Sortiert die Elemente, die die DynamicListSortable speichert
Public Sub Sort()
    Dim locÄußererZähler, locInnererZähler As Integer
    Dim locDelta As Integer
    Dim locItemTemp As flexiblerDatentyp

    locDelta = 1

    'Größten Wert der Distanzfolge ermitteln
    Do
        locDelta = 3 * locDelta + 1
    Loop Until locDelta > myCurrentCounter

    Do
        locDelta \= 3

        'Shellsort's Kernalgorithmus
        For locÄußererZähler = locDelta To myCurrentCounter - 1
            locItemTemp = Me.Item(locÄußererZähler)
            locInnererZähler = locÄußererZähler
            Do While (Me.Item(locInnererZähler - locDelta).CompareTo(locItemTemp) > 0)
                Me.Item(locInnererZähler) = Me.Item(locInnererZähler - locDelta)
                locInnererZähler = locInnererZähler - locDelta
                If (locInnererZähler <= locDelta) Then Exit Do
            Loop
            Me.Item(locInnererZähler) = locItemTemp
        Next
    Loop Until locDelta = 0
End Sub

```

Möglich wird die Implementierung eines Sortieralgorithmus erst wegen der Beschränkung auf Typen, die die `IComparable`-Schnittstelle einbinden. Doch da die Typen die Schnittstelle einbinden müssen (denn andernfalls wäre gar keine Instanzerstellung der `DynamicListSortable` möglich), kann die `Sort`-Methode auch gefahrlos auf die `CompareTo`-Methode jedes Elements zurückgreifen (siehe fett hervorgehobene Zeile im oben stehenden Listing).

Da, wie schon gesagt, beispielsweise alle primitiven Datentypen in .NET `IComparable` einbinden, können wir nun diese auch mit unserer Liste verwenden, wie der Modulcode im Folgenden zeigt:

```

Module mdlMain

    Sub Main()
        Dim locDoubleList As New DynamicListSortable(Of Double)
        locDoubleList.Add(124)
        locDoubleList.Add(1243)
        locDoubleList.Add(24)
        locDoubleList.Add(14)
    End Sub
End Module

```

```
locDoubleList.Add(1)
locDoubleList.Add(-32)
locDoubleList.Add(231)
locDoubleList.Add(143)

locDoubleList.Sort()
For Each locItem As Double In locDoubleList
    Console.WriteLine(locItem)
Next
Console.WriteLine()

Dim locStringList As New DynamicListSortable(Of String)
locStringList.Add("Klaus")
locStringList.Add("Arnold")
locStringList.Add("Sarah")
locStringList.Add("Christiane")
locStringList.Add("Jürgen")
locStringList.Add("Uta")
locStringList.Add("Helge")
locStringList.Add("Uwe")

locStringList.Sort()
For Each locItem As String In locStringList
    Console.WriteLine(locItem)
Next

Console.WriteLine()
Console.WriteLine("Taste drücken zum Beenden!")
Console.ReadKey()
End Sub

End Module
```

Auch hier sind die eigentlich interessanten Zeilen in Fettschrift gedruckt. Sie demonstrieren einerseits, dass die generische Liste auf unterschiedlichen Datentypen basieren kann und andererseits, dass dennoch solche komplexe Funktionen wie das Sortieren funktionieren – obwohl zum Zeitpunkt der Entwicklung der Liste noch gar nicht bekannt ist, mit welchen Typen es die Liste später zu tun haben wird.

Dass dieses Konzept auch funktioniert, zeigt die Ausführung des Programms, die folgendes Ergebnis ins Konsolenfenster zaubert:

```
-32
1
14
24
124
143
231
1243

Arnold
Christiane
Helge
Jürgen
```

Klaus
Sarah
Uta
Uwe

Taste drücken zum Beenden!

HINWEIS

Einen Sortieralgorithmus in eigenen Auflistungsklassen zu implementieren ist im Übrigen genau so überflüssig, wie eigene Auflistungsklassen von Grund auf neu zu kreieren. Das .NET Framework kennt nämlich eine Vielzahl von Auflistungsklassen für die unterschiedlichsten Zwecke. Doch es ist allemal interessant zu sehen, wie das Prinzip von Auflistungsklassen an sich funktioniert, und für das bessere Verständnis des vorherigen Kapitels, das die wichtigsten Auflistungen im .NET Framework vorstellt, bestimmt von Vorteil.

Beschränkungen auf Klassen, die über einen Standardkonstruktor verfügen

In einigen Fällen ist es notwendig, dass eine generische Klasse in der Lage ist, den Typ, auf dem sie basieren soll, auch zu instanziiieren. Das kann sie dann nicht, wenn es sich beim Typ, auf dem sie basiert, um eine abstrakte Klasse handelt, um eine Schnittstelle oder um eine Klasse, die ausschließlich über parametrisierte Konstruktoren verfügt.

Wenn Sie diese Fälle für den Typ ausschließen möchten, den eine generische Klasse einbindet, müssen Sie eine Beschränkung definieren, die vom Typ, auf dem sie basieren soll, einen Standardkonstruktor erfordert, und das geht folgendermaßen:

```
Public Class GenerischeKlasseMitInstanzierbaremTyp(Of flexiblerDatentyp As New)

    Public Sub TestMethode()
        'Das geht nur auf Grund der angegebenen Beschränkung:
        Dim locTest As New flexiblerDatentyp

        'Und hier ist locTest jetzt als Datentyp instanziiert!
        Console.WriteLine(locTest.ToString)
    End Sub
End Class
```

Beschränkungen auf Wertetypen

Möchten Sie eine generische Klasse auf Wertetypen beschränken, verfahren Sie auf ähnliche Weise, wie im vorherigen Abschnitt beschrieben. Sie bestimmen durch `As Structure`, dass nur noch Wertetypen (in Visual Basic also Strukturen) innerhalb einer generischen Klasse als Typ zur Anwendung kommen dürfen, die auf `ValueType` basieren. Ein Beispiel:

```
Public Class GenerischeKlasseNurMitWertetypen(Of flexiblerDatentyp As Structure)

    Public Sub TestMethode()
        'Ist Wertetyp – keine Instanzierung durch New erforderlich!
        Dim locWerteTyp As flexiblerDatentyp
    End Sub
End Class
```

```

        'Und hier ist locTest jetzt als Datentyp instanziiert.
        Console.WriteLine(locWerteTyp.ToString)
    End Sub
End Class

```

Kombinieren von Beschränkungen und Bestimmen mehrerer Typparameter

In Visual Basic sind alle Beschränkungen für generische Datentypen untereinander kombinierbar. Im Gegensatz zu Beschränkungen bei Basisdatentypen können Sie darüber hinaus auch Beschränkungen für mehrere Schnittstellen bestimmen.

Wenn Sie verschiedene Beschränkungen oder mehrere Schnittstellen für einen generischen Datentyp einrichten, fassen Sie die verschiedenen Vorschriften in geschweiften Klammern zusammen.

Ein Beispiel soll auch diesen Sachverhalt verdeutlichen:

```

Public Class GenerischeBeschränkungskombi(Of flexiblerDatentyp As {Structure, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As flexiblerDatentyp
        Dim locWerteTyp2 As flexiblerDatentyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)

        'Disposable, dank IDisposable
        locWerteTyp.Dispose()
        locWerteTyp2.Dispose()
    End Sub
End Class

```

Zusätzlich können Sie eine generische Klasse auch für die Verwendung von mehreren Typparametern einrichten. Falls Sie beispielsweise eine Auflistung entwickeln möchten, die als ein Wörterbuch fungiert, dann benötigen Sie einen Typ zum Nachschlagen (den Schlüssel) und einen für den eigentlichen Wert. (Programmieren Sie das aber nicht selbst, denn auch das gibt es schon beispielsweise mit der generischen `KeyedCollection`-Klasse, die sich im `System.Collection.ObjectModel`-Namespace befindet.) Die Einschränkungen lassen sich dann für jeden Typparameter einzeln festlegen:

```

Public Class GenerischesWörterbuch(Of Schlüsseltyp As {Structure, IComparable}, _
                                   Wertetyp As {New, IComparable, IDisposable})

    Public Sub TestMethode()
        Dim locWerteTyp As Schlüsseltyp
        Dim locWerteTyp2 As Wertetyp

        'Direkt verwendbar, da Wertetyp durch Struktur
        'Vergleichbar, dank IComparable
        locWerteTyp.CompareTo(locWerteTyp2)
    End Sub
End Class

```

```

        'Disposable, dank IDisposable
        locWerteTyp2.Dispose()
    End Sub
End Class

```

Generische Auflistungen (Generic Collections)

Generische Auflistungen haben im Vergleich zu »herkömmlichen« Auflistungen, die Sie im vorherigen Kapitel kennen gelernt haben, einen entscheidenden Vorteil: Sie sind grundsätzlich typsicher. Anders als »normale« Auflistungen wie beispielsweise die `ArrayList`-Klasse nehmen sie, da sie nicht auf `Object` basieren, nicht jeden Datentyp als Element entgegen, sondern beschränken sich auf den Datentyp, der ihrer Definition zugrunde liegt.

Das bedeutet: Definieren Sie beispielsweise eine `Collection` auf Basis von `Integer`, etwa mit

```
Dim locGenColl As New Collection(Of Integer)
```

dann laufen Sie nicht Gefahr, später versehentlich der Auflistung ein Element hinzuzufügen, das nicht vom Typ `Integer` ist – oder mit anderen Worten: Die Zeile

```
locGenColl.Add("Ein Element")
```

würde bereits zur Entwurfszeit im Editor als fehlerhaft gekennzeichnet.

Ihre Programme werden durch den Einsatz von generischen Auflistungen somit robuster und auch die Entwicklungszeit reduziert sich, da Sie sich nicht mit Laufzeitfehlern herumärgern müssen, sondern bereits zur Entwurfszeit Fehler korrigieren können. Und weniger Programmtests bedeuten weniger Entwicklungszeit und -kosten.

Nun gibt es nicht wenige generische Auflistungen seit dem .NET Framework 2.0, und sie alle im Detail zu beschreiben ist nicht nur unnötig – schließlich funktionieren viele von ihnen wie ihre nicht generischen Verwandten – es würde auch den Rahmen des Buches sprengen.

Aus diesem Grund möchte ich mich auf die in meinen Augen wichtigen Besonderheiten beschränken, die Sie bei nicht-generischen Auflistungen nicht antreffen. Eine Tabelle, die Sie im Folgenden finden, gibt darüber hinaus Auskunft, welche generischen Auflistungen Ihnen zur Verfügung stehen, und zu welchem Zweck sie dienen.

Namespace	Auflistung	Beschreibung
System.Collection.ObjectModel	Collection(Of type)	Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung. Besonderheiten: Im Gegensatz zu <code>List(Of type)</code> gibt es überschreibbare Methoden, mit denen man das Verhalten beim Einfügen, Löschen und Neuzuweisen von Elementen der Auflistung in abgeleiteten Klassen beeinflussen kann. ▶

Namespace	Auflistung	Beschreibung
System.Collection.ObjectModel	KeyedCollection(Of key, type)	Stellt eine Auflistung von Elementen zur Verfügung, die sowohl das Nachschlagen über einen Schlüssel (Key) eines bestimmten Typs <i>sowie</i> den Einsatz eines Indexes erlaubt. Besonderheiten: Diese Auflistung können Sie nur in Ableitungen verwenden, da der Typ, den Sie speichern, selber einen Standard-Schlüssel erzeugen muss, und für diesen Umstand müssen Sie in der Ableitung sorgen. WICHTIG: Vermeiden Sie den Einsatz von numerischen Schlüssel (Integer, Long, etc.), da es hier beim Serialisieren der Auflistung zu Problemen kommt (Stand: <i>.NET Framework-Version 2.0.50727</i>).
System.Collection.ObjectModel	ReadOnlyCollection(Of type)	Stellt eine Auflistung dar, deren Elemente nur gelesen werden können. Besonderheiten: Elemente, die in einer anderen generischen Auflistung gleichen Typs vorliegen, können nur bei der Instanziierung im Konstruktor dieser Auflistung übergeben werden. Ansonsten sind die Elemente nur lesbar und können nach der Instanziierung nicht mehr verändert werden.
System.Collections.Generic	Dictionary(Of key, type)	Stellt eine Auflistung von Schlüssel und Werten dar. Besonderheiten: Stellt eine Zuordnung von einem Satz von Schlüssel zu einem Satz von Werten bereit. Jede Hinzufügung zum Wörterbuch besteht aus einem Wert und dem zugeordneten Schlüssel. Ein Wert kann anhand des zugehörigen Schlüssel sehr schnell abgerufen werden (beinahe ein O(1)-Vorgang), da die Dictionary-Klasse in Form einer Hashtable implementiert ist.
System.Collections.Generic	LinkedList(Of type)	Stellt eine doppelt verknüpfte Liste dar. Besonderheiten: Ist eine verknüpfte Liste mit einzelnen Knoten vom Typ <code>LinkedListNode</code> ; das Einfügen und Entfernen einzelner Elemente geht extrem schnell vonstatten.
System.Collections.Generic	List(Of type)	Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung. Hinweis: Diese Klasse eignet sich nicht für Ableitungen in eigenen Auflistungsklassen, bei denen Sie mit Code Einfluss auf die Bearbeitung der Liste nehmen müssen. Verwenden Sie stattdessen die <code>Collection(Of)</code> -Auflistung (siehe oben).
System.Collections.Generic	Queue(Of type)	Stellt eine FIFO-Auflistung (First-In-First-Out) von Objekten dar. Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel.
System.Collections.Generic	SortedDictionary(Of key, type)	Stellt eine Auflistung von Schlüssel-Wert-Paaren dar, deren Reihenfolge anhand des Schlüssel bestimmt wird. ▶

Namespace	Auflistung	Beschreibung
System.Collections.Generic	SortedList(Of key, type)	Verwaltet eine sortierte Liste, deren Elemente über Schlüssel abrufbar sind. Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel. Im Gegensatz zu SortedDictionary erfolgt die Sortierung über das Element und nicht über den Schlüssel!
System.Collections.Generic	Stack(Of type)	Stellt eine LIFO-Auflistung (Last-In-First-Out) von Objekten dar. Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung finden Sie im vorherigen Kapitel.

Tabelle 6.1: Die wichtigsten generischen Auflistungstypen

List(Of)-Auflistungen und Lambda-Ausdrücke

Die List(Of)-Klasse ist eine der am häufigsten (wenn nicht sogar die am häufigsten) eingesetzte generische Auflistungsklasse. Wenn Sie primitive Datentypen in einer Auflistung speichern, sollten Sie den Einsatz List(Of type) anderen Auflistungsklassen vorziehen – alleine schon aus Performance-Gründen. Das .NET Framework ist nämlich in der Lage, den eigentlich notwendigen Boxing-Vorgang bei List(Of type) zu umgehen, und damit ist List(Of type) die schnellere Alternative.

Und obwohl es sich bei dieser generischen Auflistung um solch eine populäre Klasse handelt, hält sie Funktionalitäten bereit, über die man – obwohl sie bereits im .NET Framework 2.0 vorhanden waren – jedenfalls im Rahmen von Visual Basic bislang wenig Brauchbares erfahren konnte; wohl auch, weil sie förmlich nach der Verwendung von Lambda-Funktionen schreien, die erst mit der 2008er Version Einzug in Visual Basic hielten. Das folgende Beispiel setzt das Verständnis von Lambda-Funktionen voraus – und falls Sie sich mit diesen noch nicht auseinander gesetzt haben sollten: Kapitel 3 klärt deren grundsätzliches Verständnis.

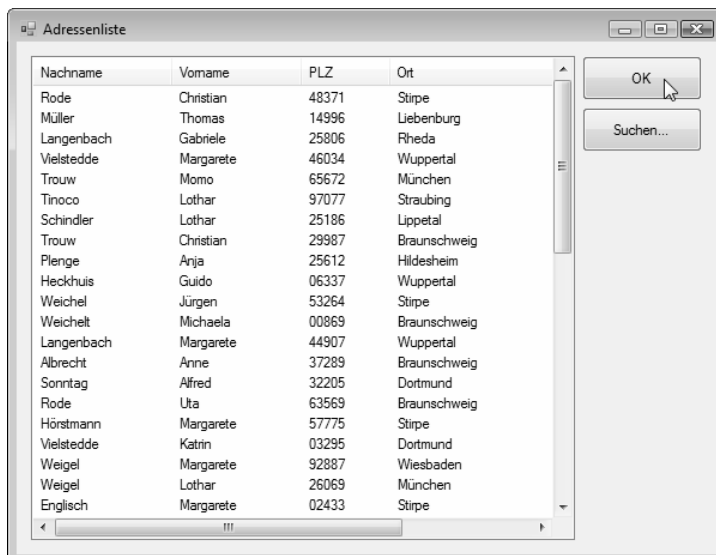


Abbildung 6.7 Sortierung und Suche steuern Sie über die Spaltenköpfe in dieser Adressenliste

BEGLEITDATEIEN

Sie finden das folgende Beispielprojekt im Verzeichnis `.\Samples\Chapter06 - Generics\ListOfDemo`.

Wenn Sie das Programm starten, sehen Sie einen Dialog, etwa wie in Abbildung 6.7 zu sehen. Die im Formular vorhandene `Listview` wird mit 50 Zufallsadressen gefüllt. Sie können die Liste sortieren, indem Sie auf den entsprechenden Spaltenkopf klicken – so, wie Sie es vom Windows-Explorer in der *Details*-Ansicht gewohnt sind.

Eine Suchfunktion, die Sie über die entsprechende Schaltfläche erreichen, gestattet es Ihnen, nach dem Begriff innerhalb der Liste zu suchen. Gefunden wird dabei der Eintrag, dessen Text in der Spalte, nach der zuletzt sortiert wurde, dem eingegebenen Suchbegriff entspricht.

Soweit ist dieses Beispiel noch nichts Besonderes. Doch wenn Sie einen Blick in die Umsetzung riskieren, wird klar, wie alternativ der Lösungsansatz ist. Das geht schon beim Schreiben der Zufallsadressen in die Liste los:

ForEach und die generische Action-Klasse

Wenn Sie den Inhalt eines Arrays oder einer Auflistung in einer Liste darstellen möchten, liegt die Vorgehensweise, um das zu erreichen, eigentlich auf der Hand: Sie iterieren mit einem `For/Each`-Konstrukt durch die Auflistung, verarbeiten damit jedes Element und bringen es mit geeigneten Methoden oder Eigenschaften in die sichtbare Liste eines Formulars.

Dieses Beispiel nutzt einen anderen Weg, wie im folgenden Listing-Ausschnitt zu sehen:

```
Sub ElementeDarstellen()  
  
    'Unterdrückt Neuzeichnen-Ereignisse bis zum  
    'nächsten EndUpdate; dadurch geht der Aufbau  
    'der Elemente schneller und wackelt nicht.  
    Me.lvwAdressen.BeginUpdate()  
  
    'Alle Elemente der ListView löschen.  
    Me.lvwAdressen.Items.Clear()  
  
    'Für jedes Element der Liste wird ElementInListe aufgerufen.  
    myAdressen.ForEach(New Action(Of Adresse) (AddressOf ElementInListe))  
  
    'So werden die Spaltenbreiten optimal angepasst.  
    For Each locCol As ColumnHeader In Me.lvwAdressen.Columns  
        locCol.Width = -2  
    Next  
  
    'Aufbau der ListView ist beendet.  
    Me.lvwAdressen.EndUpdate()  
End Sub  
  
'Der Action-Delegat: für jedes Element der Liste wird diese Aktion durchgeführt.  
Sub ElementInListe(ByVal Element As Adresse)  
    'Neues ListView-Element - Name kommt zuerst.  
    Dim locLvwItem As New ListViewItem(Element.Name)  
  
    'Die Untereinträge setzen  
    With locLvwItem.SubItems
```

```

        .Add(Element.Vorname)
        .Add(Element.PLZ)
        .Add(Element.Ort)
    End With

    'Zum Wiederfinden Referenz in Tag
    1ocLvwItem.Tag = Element

    'Zur Listview hinzufügen
    1vwAdressen.Items.Add(1ocLvwItem)
End Sub

```

Sie können die `ForEach`-Methode verwenden, um durch eine Auflistung iterieren *zu lassen* und dabei für jedes Element einen Delegaten aufrufen, der eine bestimmte Aktion ausführt. Genau das Verfahren nutzen wir an dieser Stelle, um die Liste aufzubauen. Der Delegat wird im Beispiel nicht durch ein `Delegate`-Objekt (mehr zu Delegaten finden Sie in Kapitel 4) sondern durch die direkte Angabe einer Prozedur mithilfe des `Address Of`-Operators angegeben – der Compiler sorgt dann für den richtigen Ersatz einer Delegatvariablen. Aber natürlich könnten an dieser Stelle auch »manuelle« Delegatvariablen zum Einsatz kommen, die in Abhängigkeit von bestimmten Programmzuständen andere Prozeduren verwenden, und genau darin besteht der Reiz des Einsatzes dieser Verfahren. Und – auch das ist möglich – mit Visual Basic 2008 können Sie anstelle eines Delegaten auch einen Lambda-Ausdruck an dieser Stelle einsetzen.

Wichtig ist in jedem Fall, dass die Routinen, die Sie mithilfe der `Action`-Klasse aufrufen, den Signaturanspruch des Delegaten erfüllen, den Sie im Konstruktor von `Action` angeben. Im Fall von `ForEach` und der `Action`-Klasse muss es sich bei der Delegatenprozedur um eine Methode handeln, die kein Funktionsergebnis hat (also um eine Visual Basic-Sub) und als Parameter ein Element entgegen nimmt, dessen Typ auch der Basis der generischen `Action`-Klasse entspricht – Adresse in unserem Beispiel.

Im Ergebnis erreichen wir also, dass für jedes Element des `List(Of Adresse)`-Objektes `myAdressen` die Methode `ElementInListe` aufgerufen wird.

Sort und die generische Comparison-Klasse

Prinzipiell funktioniert das nächste Pärchen ähnlich, das Sie verwenden, wenn Sie ein Array mit der Methode `Sort` ohne den Einsatz einer speziellen `Comparer`-Klasse (wie in Kapitel 5 gezeigt) sortieren möchten. In Abwandlung zur ersten Verwendung kommen hier jedoch Lambda-Ausdrücke zum Einsatz, sodass wir uns das buchstäbliche Delegieren an die richtige Sortierungsfunktion mit einer Delegatenvariable sparen können. Den relevanten Codeausschnitt des Beispiels finden Sie im Folgenden:

```

'Wird aufgerufen, wenn eine der Spalten angeklickt wird.
Private Sub 1vwAdressen_ColumnClick(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.ColumnClickEventArgs) Handles 1vwAdressen.ColumnClick

    'Spaltennummer, die in e.Column steht, in AdressenSortierenNach konvertieren
    mySortierenNach = CType(e.Column, AdressenSortierenNach)

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
                String.Compare(adr1.Name, adr2.Name))
    End Select
End Sub

```

```

Case AdressenSortierenNach.Vorname
    myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
        String.Compare(adr1.Vorname, adr2.Vorname))

Case AdressenSortierenNach.PLZ
    myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
        String.Compare(adr1.PLZ, adr2.PLZ))

Case AdressenSortierenNach.Ort
    myAdressen.Sort(Function(adr1 As Adresse, adr2 As Adresse) _
        String.Compare(adr1.Ort, adr2.Ort))

End Select

'Die Elemente neu sortiert darstellen
ElementeDarstellen()
End Sub

```

Sort arbeitet hier in diesem Beispiel mit Lambda-Ausdrücken, bei denen jeder der vier Lambda-Ausdrücke nach einer anderen Eigenschaft der Adresse sortiert – in Abhängigkeit davon, welche der vier Spalten in der ListView zum Sortieren eingestellt wurde.

Find und die generische Predicate-Klasse

Das letzte generische »Dreamteam« schließlich kommt zum Einsatz, wenn es um das Finden eines bestimmten Objektes in einer generischen Liste geht: Find und der Delegat, der durch die generische Predicate-Klasse eingerichtet wird. Auch hier entspricht die grundsätzliche Vorgehensweise dem bereits Bekannten, wie der entsprechende Code im Beispiel zeigt:

```

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

    'Suchbegriff abfragen
    Dim suchFormular As New frmSuchen
    Dim aktuellerSuchbegriff As String ' Der zuletzt erfragte Suchbegriff

    'Den Suchbegriff merken, damit der Predicate-Delegat darauf
    'zugreifen kann.
    aktuellerSuchbegriff = suchFormular.Suchbegriff
    If String.IsNullOrEmpty(aktuellerSuchbegriff) Then
        Return
    End If

    'Hier kann die Suche beginnen!
    Dim locGefAdr As Adresse = Nothing

    Select Case mySortierenNach
        Case AdressenSortierenNach.Name
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Name = aktuellerSuchbegriff)
        Case AdressenSortierenNach.Vorname
            locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Vorname = aktuellerSuchbegriff)
    End Select

```

```

Case AdressenSortierenNach.PLZ
    locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.PLZ = aktuellerSuchbegriff)
Case AdressenSortierenNach.Ort
    locGefAdr = myAdressen.Find(Function(adr As Adresse) adr.Ort = aktuellerSuchbegriff)
End Select

'Wenn ein Element gefunden wurde, dieses markieren.
If locGefAdr IsNot Nothing Then

    'Alle ListView-Elemente durchsuchen und überprüfen, ob ...
    For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

        '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
        If locLvwItem.Tag Is locGefAdr Then

            'Gefunden! ListView-Element markieren,
            locLvwItem.Selected = True

            'und dafür sorgen, dass es im sichtbaren Bereich liegt.
            locLvwItem.EnsureVisible()
            Return
        End If
    Next
End If
End Sub

```

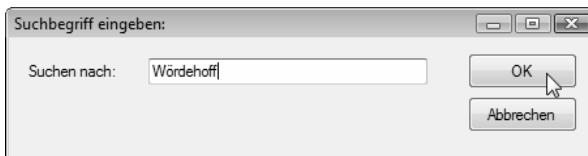


Abbildung 6.8 Der Suchbegriff bezieht sich immer auf die Spalte, nach der zuletzt sortiert wurde

Auch hier machen Lambda-Ausdrücke, wie in den fetthervorgehobenen Codezeilen zu sehen, die Suche nach den richtigen Adressen vergleichsweise einfach. Je nach selektierter Sortierspalte wird einfach ein anderer Lambda-Ausdruck verwendet, der den Vergleich mit dem eingegebenen Suchbegriff übernimmt.

Nur eine kleine Herausforderung ist dann noch das Selektieren des gefundenen Begriffs in der Liste – und das ist im Übrigen auch der Grund, wieso wir beim Aufbau der Liste eine Referenz jedes Elements in der Tag-Eigenschaft jedes ListViewItem-Elements speichern: Wenn der Begriff durch Find gefunden wurde, liegt uns das entsprechende Adresse-Objekt vor. Mit diesem können wir anschließend durch die ListViewItem-Auflistung iterieren und auf Objektübereinstimmung durch Abfrage der Tag-Eigenschaft testen. Dieser Aufwand ist nötig, da es keine andere Möglichkeit gibt, das richtige ListViewItem-Element zu finden, und nur dieses erlaubt es aber, die richtige Zeile in der Liste durch seine Select-Eigenschaft zu selektieren.