

14 Generische Klassen und Strukturen (Generics)

-
- 393 Verwenden einer Codebasis für verschiedene Typen
 - 394 Lösungsansätze
 - 397 Typengeneralisierung durch den Einsatz generischer Datentypen
 - 400 Beschränkungen (Constraints)
 - 410 Vererben von generischen Typen
-

Verwenden einer Codebasis für verschiedene Typen

Wenn Sie bestimmte Klassen oder Strukturen, also wie auch immer geartete Typen, entwickeln, haben diese unter Umständen einen Nachteil: Sie verarbeiten, wenn sie typsicher 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 typsicher und kann zur Laufzeit Ausnahmen auslösen, mit denen Sie nicht rechnen.

BEGLEITDATEIEN: Ein Beispiel, das Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap14\Generics01` finden, soll das verdeutlichen.

Dieses Beispiel verwendet die aus ► Kapitel 9 bereits 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:

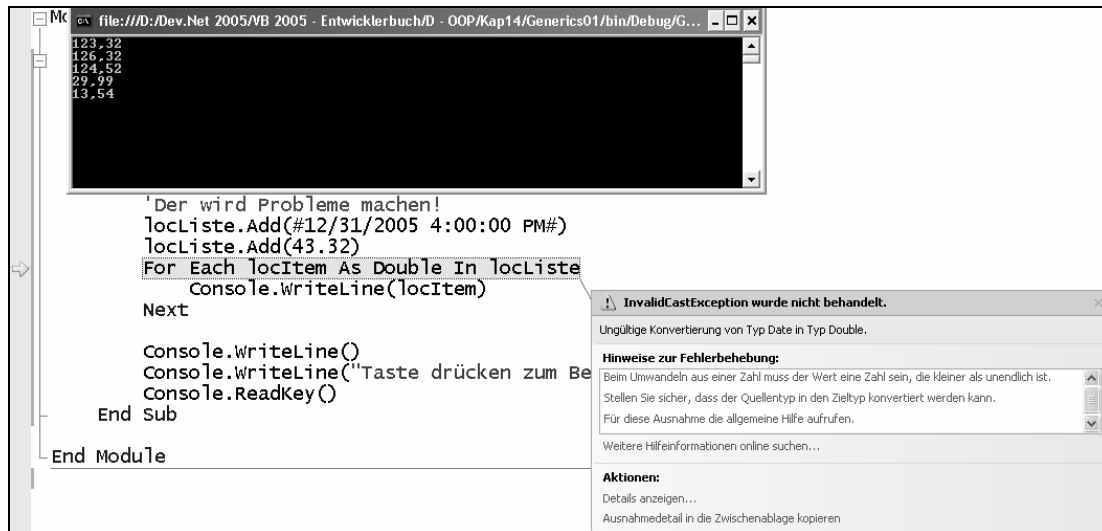


Abbildung 14.1: Die Liste wird nur bis zum *Date* Element ausgegeben; das *Date*-Element ist nämlich nicht in *Double* konvertierbar

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 *DynamicList-*

Double (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, 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
```

```
        'Elemente kopieren; das geht mit dieser  
        'statischen Methode extrem schnell, da zum Einen nur die  
        'Zeiger kopiert werden, zum anderen diese Routine  
        'intern nicht in Managed Code sondern nativem Assembler ausgeführt wird.  
        Array.Copy(myArray, locTempArray, myArray.Length)
```

```
        'Auch hier werden nur die Zeiger auf die Elemente "verbogen".  
        'Die vorherige Liste der Zeiger in myArray, die nun verwaist ist,  
        'fällt dem Garbage Collector zum Opfer.  
        myArray = locTempArray
```

End If

End Sub

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

```

    End Get
End Property

'Erlaubt das Zuweisen
Default Public Overridable Property Item(ByVal Index As Integer) As Double
    Get
        Return myArray(Index)
    End Get

    Set(ByVal Value As Double)
        myArray(Index) = Value
    End Set
End Property

Public Function GetEnumerator() As System.Collections.IEnumerator _
    Implements System.Collections.IEnumerable.GetEnumerator
    'HINWEIS: GetEnumerator muss implementiert werden, wenn die
    'IEnumerable-Schnittstelle eingebunden wird. Die wiederum
    'ist notwendig, um For/Each zu ermöglichen.
    'Ein Array ist Enumarable - deswegen klauen wir uns
    'dessen GetEnumerator-Funktionalität.
    'Wir müssen bloß ein lokales Array erstellen,
    'das genau so viele Elemente hat, wie sich derzeit
    'in der DynamicList-Instanz befinden, damit die noch
    'nicht genutzten Array-Elemente auch noch aufgezählt werden.
    'Aus diesem Grund verwenden wir eine lokale Kopie,
    'die exakt so groß ist, wie wir Anzahl Elemente
    'in dieser Liste haben.
    Dim locTempArray(myCurrentArraySize) As Double
    Array.Copy(myArray, locTempArray, myArray.Length)
    Return myArray.GetEnumerator
End Function
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 auch in Abbildung 14.2 sehen können.

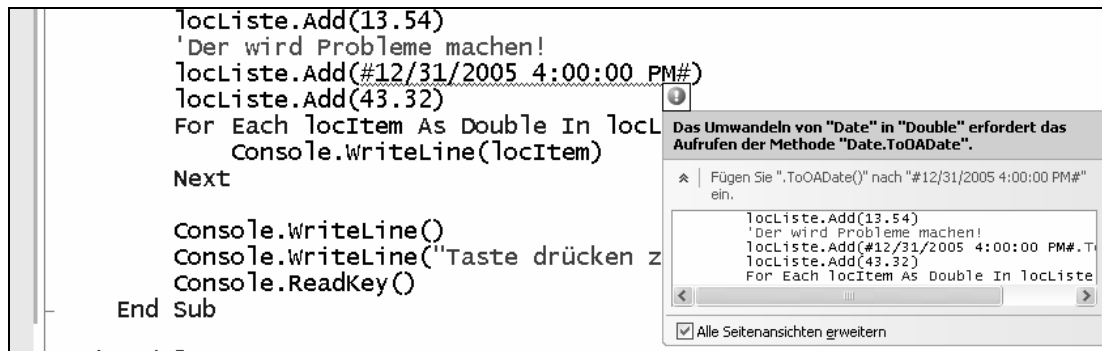


Abbildung 14.2: Bei typsicheren Klassen sehen Sie Fehlermeldungen bei »falschen« Typen bereits im Editor zur Entwurfszeit – wengleich die Fehlermeldungen ab und an über das Ziel hinausschießen

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 in Ihrer späteren Entwicklerzeit 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 Basis-Klasse 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 googeln 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 ersetzt¹.

BEGLEITDATEIEN: Das folgende Beispiel finden Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap14\Generics02`.

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 diese Typparameter als Stellvertreter ein. Für unsere `DynamicList`-Klasse bedeutet das:

```
Class DynamicList(Of flexiblerDatentyp)
    Implements IEnumerable

    Protected myStep As Integer = 4          ' Schrittweite, 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
```

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. Wenn Sie sich für die interne Funktionsweise von Generics interessieren – der IntelliLink *D1401* hält Interessantes zum Thema bereit!

```

        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
    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 14.3 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)
```

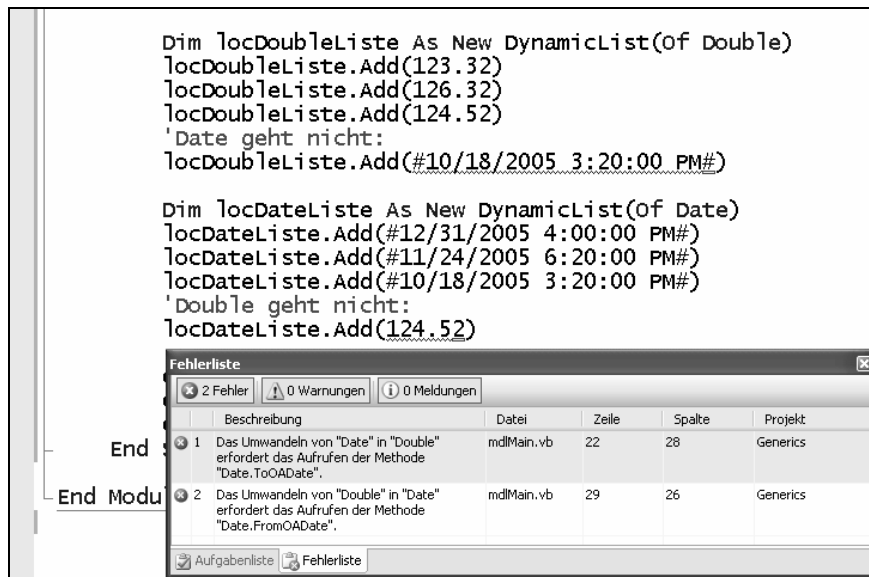


Abbildung 14.3: Mit *Of* bestimmen Sie, für welchen Datentyp Ihre generische Klasse zur Anwendung kommen soll

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.

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.

Beschränkungen für generische Typen auf eine bestimmte Basisklasse

Dazu ein Beispiel: Angenommen, Sie haben eine Anwendung geschaffen, die die verschiedensten Körper (Quader, Kugel, 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 die `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
        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 Körpers 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
```

```

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 die folgende Abbildung zeigt:

```

Module mdMain
    Sub Main()
        Dim lockörperliste As New DynamicList(Of KörperBasis)
        With lockörperliste
            .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:" & _
            lockörperliste.Gesamtvolumen)
    End Sub
End Module

```

Abbildung 14.4: 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 14.4 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 trägt, und deren `Volumen`-Eigenschaft abfragt. Doch genau eine solche Prozedur können wir nicht implementieren, wie die folgende Grafik zeigt:

```

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

```

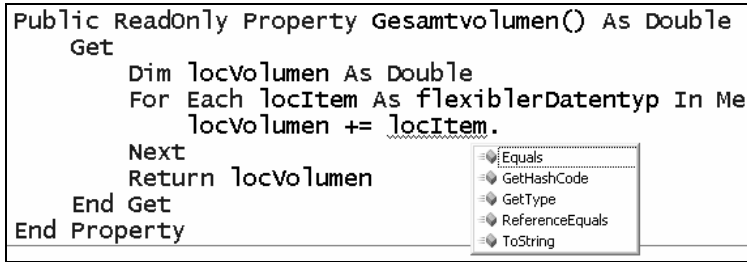


Abbildung 14.5: Die Eigenschaft, die zur Berechnung des Gesamtvolumens benötigt wird, ist nicht erreichbar!

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 zur Verfügung gestellt werden. Das sind logischerweise genau die Eigenschaften und Methoden, die `Object` zur Verfügung stellt, denn nur diese erbt, wie wir ja schon wissen, automatisch jede neue Klasse implizit.

Und genau das ist 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 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, 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: Neben der Erweiterung (Of `flexiblerDatentyp`, die den Typparameter für die weitere Verwendung des generischen Typs in der Klasse festlegt, gibt es obendrein den 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 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 locKörperliste As New DynamicListKörper(Of KörperBasis)
        With locKörperliste
            .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:" & _
            locKörperliste.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.

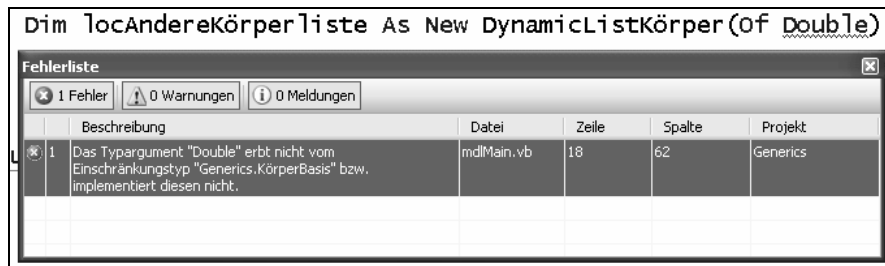


Abbildung 14.6: Eine beschränkte generische Klasse können Sie tatsächlich nur noch auf Basis eines Datentyps instanzieren, der die Beschränkung erfüllt

HINWEIS: Das Framework erlaubt es nicht, mehrere Basisklassen als Beschränkung für einen generischen Typ zu definieren. Das würde die Technik der Mehrfachvererbung implizieren, die das 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 diese in ihren Beschränkungen nicht auf bestimmte Basisklassen, sondern nur auf bestimmte Schnittstellen beschränken.

Darüber hinaus haben Schnittstellen den Vorteil, sich bereits in vielen »fertigen« Typen des 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 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 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 `.\VB 2005 - Entwicklerbuch\D - OOP\Kap14\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, 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 locÄuBererZähler, 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
            'War einen zu groß, also wieder teilen
            locDelta \= 3

            'Shellsort's Kernalgorithmus
            For locÄuBererZähler = locDelta To myCurrentCounter - 1
                locItemTemp = Me.Item(locÄuBererZähler)
                locInnererZähler = locÄuBererZähler
                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 anderenfalls gar keine Instanzerstellung der DynamicListSortable möglich wäre), 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)
    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 Zeilen in Fettschrift die eigentlich interessanten. Sie demonstrieren einerseits, dass die generische Liste auf unterschiedlichen Datentypen basieren kann und andererseits, dass dennoch solch 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 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 von ► Kapitel 19, 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 instanzieren. 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 GenerischeKlasseMitInstanzierbarenTyp(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 GenerischeKlasseNurMitWertetyp(Of flexiblerDatentyp As Structure)

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

        '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}, _
    Werttyp As {New, IComparable, IDisposable})

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

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

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

```

Vererben von generischen Typen

Generische Datentypen lassen sich natürlich auch vererben. Dabei können Sie bestimmen, ob Sie den Typparameter auflösen, und die vererbte Klasse auf einen bestimmten Datentypen beschränken, oder auch die vererbte Klasse generisch anlegen und sie damit wieder flexibel für den Einsatz mit beliebigen Datentypen machen.

Eigentlich ist das, was wir im ► Abschnitt »Beschränkungen für generische Typen auf eine bestimmte Basisklasse« ab Seite 393 gemacht haben, furchtbar uneffizient. Wir haben den Quellcode mit *Kopieren/Einfügen* kopiert – und das widerspricht nun wirklich jedem Grundsatz der objektorientierten Programmierung.

Viel besser wäre es gewesen, die `DynamicList`-Klasse zu vererben, und die Ableitung dieser Klasse so zu modifizieren, dass sie den geforderten Ansprüchen gerecht geworden wäre. Das wäre nicht nur weniger Arbeit gewesen; wenn Sie im Nachhinein einen Fehler in der Basisklasse entdecken und diesen beheben, korrigieren sie ihn zwangsläufig auch in jeder Ableitung.

Möchten wir das für unsere `DynamicList` und die `KörperBasis`-Klasse machen, können wir uns prinzipiell entscheiden: Lösen wir für eine neue `DynamicList`-Klasse, die nur Objekte auf Basis von `KörperBasis` akzeptiert, die generische Eigenschaft auf, oder belassen wir sie generisch.

In diesem Fall macht es keinen Sinn, die Klasse weiterhin generisch zu »halten«, denn: Da die `DynamicList` mit der Beschränkung auf `KörperBasis`-Objekte sowieso keine anderen Objekte mehr akzeptiert, können wir in der Ableitung ihre generische Eigenschaft auch auflösen. Eine solche Implementierung würde folgendermaßen ausschauen:

```

Class Körperliste
    Inherits DynamicList(Of KörperBasis)

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

```

```
End Get
End Property
End Class
```

In diesem Fall lösen wir also die generische Eigenschaft der `DynamicList` auf und überführen die alte generische Klasse in eine neue nicht generische Klasse namens `KörperList`. Die Auflösung geschieht dadurch, dass der Typ, den die geerbte Klasse ab sofort verwalten soll, in der `Inherits`-Anweisung angegeben wird.

Die Alternative dazu, die, wie gesagt, für unser Beispiel nicht wirklich Sinn ergeben würde, und bei der ihre generische Eigenschaft erhalten bliebe, sähe folgendermaßen aus:

```
Class KörperlisteImmerNochGenerisch(Of flexiblerDatentyp As KörperBasis)  
Inherits DynamicList(Of flexiblerDatentyp)
```

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

Diese Art der Vererbung macht nur dann Sinn, wenn die Beschränkung, eben nicht wie in diesem Beispiel, so restriktiv ist, dass sich die Klasse ohnehin nur noch auf einen Typen (und dessen Ableitungen) beschränkt – wenn die Ausgangsklasse also entweder mit *keiner* Beschränkung, mit *mehreren* Schnittstellen oder mit der `New`- oder `Structure`-Beschränkung arbeitet.

