

20 Arbeiten mit generischen Typen und generischen Auflistungen

-
- 595 Wertetypen, die Nothing speichern können – Nullable(Of)
 - 599 Generische Auflistungen (Generic Collections)
 - 607 Auflistungen und Aktionen (Actions), Aussageprüfer (Predicates) und Vergleiche (Comparisons)
-

Wie Sie generische Typen entwickeln, haben Sie in ► Kapitel 14 bereits kennen gelernt. ► Kapitel 16 hat – wenn auch in einem anderen Zusammenhang – den Einsatz von generischen Typen das erste Mal demonstriert. Dort konnten Sie auch schon sehen, dass gerade was Auflistungen anbelangt, es viele generische Datentypen bereits zur direkten Verwendung im .NET-Framework gibt. Dieses Kapitel gibt Ihnen einen Überblick über die wichtigsten generischen Datentypen und Auflistungen.

Wertetypen, die Nothing speichern können – Nullable(Of)

Kaum eine Neuerung hat für so viel Wirbel, Spekulationen und auch Stress in den eigenen (Entwickler-)Reihen gesorgt, wie der Datentyp `Nullable`, von dem in diesem Abschnitt die Rede sein wird.

`Nullable` ist ein generischer Datentyp mit einer Einschränkung auf Wertetypen. Er ermöglicht, dass ein beliebiger Wertetyp neben seiner eigentlichen Wertart einen weiteren Zustand »speichern« kann – nämlich `Nothing`.

Ist das wichtig? Oh ja! Beispielsweise in der Datenbankprogrammierung. Wenn Sie bereits Erfahrungen in der Datenbankprogrammierung haben, wissen Sie auch sicherlich, dass Ihre Datenbanktabellen über Datenfelder verfügen können, die den »Wert« `Null` »speichern« können – als Zeichen dafür, dass eben nichts in diesem Feld gespeichert wurde.

Ein anderes Beispiel sind `CheckBox`-Steuerelemente in `Windows Forms`-Anwendungen: Sie verfügen über einen Zwischenzustand, der den Zustand »nicht definiert« anzeigen soll. Eine einfache boolesche Variable könnte alle möglichen Zustände nicht aufnehmen – `True` und `False` sind dafür einfach zu wenig. Anders ist es, wenn Sie eine Variable vom Typ `Nullable(Of Boolean)` definieren würden. In diesem Fall könnte man eine Fallunterscheidung zwischen allen möglichen Zuständen folgendermaßen realisieren:



Abbildung 20.1: Ein `Nullable(Of Boolean)` eignet sich beispielsweise dazu, auch Zwischenzustände eines `CheckBox`-Steuerelements – wie hier im Bild zu sehen – zu speichern

BEGLEITDATEIEN: Sie finden das folgende Beispielprojekt unter `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap20\NullableUndCheckbox\`.

Dieses Beispiel demonstriert, wie alle Zustände eines `CheckBox`-Steuerelements, dessen `ThreeState`-Eigenschaft zur Anzeige aller *drei* Zustände auf `True` gesetzt wurde, in einer Member-Variablen vom Typ `Nullable(Of Boolean)` gespeichert werden können. Klicken Sie auf *Zustand speichern*, um den Zustand des `CheckBox`-Steuerelements in der Member-Variablen zu sichern, verändern Sie anschließend den Zustand, und stellen Sie den ursprünglichen Zustand des `CheckBox`-Steuerelements mit der entsprechenden Schaltfläche wieder her.

Der entsprechende Code dazu lautet folgendermaßen:

```
Public Class Form1
    Private myCheckBoxZustand As Nullable(Of Boolean)

    Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click
        Me.Close()
    End Sub

    Private Sub btnSpeichern_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnSpeichern.Click
        If chkDemo.CheckState = CheckState.Indeterminate Then
            myCheckBoxZustand = Nothing
        Else
            myCheckBoxZustand = chkDemo.Checked
        End If
    End Sub

    Private Sub btnWiederherstellen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnWiederherstellen.Click
        If Not myCheckBoxZustand.HasValue Then
            chkDemo.CheckState = CheckState.Indeterminate
        Else
            If myCheckBoxZustand.Value Then
                chkDemo.CheckState = CheckState.Checked
            Else
                chkDemo.CheckState = CheckState.Unchecked
            End If
        End If
    End Sub
End Class
```

Die Zeilen, in denen die Member-Variablen `Nullable(Of Boolean)` zum Einsatz kommen, sind im Listing fett markiert. Dabei fällt Folgendes auf:

- **Wertzuweisung:** Wenn Sie einen Wert des zugrunde liegenden Typs an `Nullable(Of)` zuweisen wollen, können Sie die implizite Konvertierung verwenden, den entsprechenden Wert also direkt zuweisen, etwa wie in der Zeile

```
myCheckBoxZustand = chkDemo.Checked
```

zu sehen.

- **Auf `Nothing` zurücksetzen:** Möchten Sie eine `Nullable`-Instanz auf `Nothing` zurücksetzen, weisen Sie ihr einfach den »Wert« `Nothing` zu – wie im Listing an dieser Stelle zu sehen:

```
myCheckBoxZustand = Nothing
```

- **Auf Wert prüfen:** Möchten Sie wissen, ob eine `Nullable`-Instanz einen Wert oder `Nothing` enthält, verwenden Sie deren Eigenschaft `HasValue`. Auch dafür gibt es ein Beispiel im Listing:

```
If Not myCheckBoxZustand.HasValue Then
    chkDemo.CheckState = CheckState.Indeterminate
Else
    .
    .
    .
```

- **Wert abrufen:** Und schließlich müssen Sie natürlich auch den Wert, den eine `Nullable`-Instanz trägt, wenn sie nicht `Nothing` ist, ermitteln können. Dazu dient die Eigenschaft `Value`. Ein Beispiel dafür:

```
If myCheckBoxZustand.Value Then
    chkDemo.CheckState = CheckState.Checked
Else
    chkDemo.CheckState = CheckState.Unchecked
End If
.
.
.
```

HINWEIS: Erst in diesem Beispiel fiel auf, dass man offensichtlich ein `CheckBox`-Steuerelement, dessen `ThreeState`-Eigenschaft gesetzt ist und das den *Intermediate*-Zustand momentan trägt, nicht mit seiner `Checked`-Eigenschaft in einen anderen Zustand versetzen kann (`Checked` oder `Unchecked`). Sie können in diesem Fall nur die `CheckState`-Eigenschaft verwenden, um das `CheckBox`-Steuerelement programmgesteuert wieder aus dem *Intermediate*-Zustand herauszuholen!

Besonderheiten bei `Nullable(Of)` beim Boxen

Der Datentyp `Nullable(Of)` ist das, was man in Visual Basic als Struktur programmieren würde, also ein Wertetyp. Doch Sie könnten diesen Wertetyp nicht 1:1 nachprogrammieren, denn er erfährt durch die Common Language Runtime eine besondere Behandlung – und das ist auch gut so.

BEGLEITDATEIEN: Das folgende Beispiel finden Sie im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap20\NullableDemo\`.

Wenn Sie eine Instanz einer beliebigen Struktur – also eines beliebigen Wertetyps – verarbeiten, kommt irgendwann der Zeitpunkt, zu dem Sie diesen Wertetyp in einer Objektvariablen boxen müssen – beispielsweise wenn Sie ihn als Bestandteil eines Arrays oder einer Auflistung (Collection) speichern.

Wann immer Sie einen definierten Wertetyp in einem Objekt boxen, kann dieses Objekt logischerweise nicht `Nothing` sein, ganz egal, welchen »Wert« diese Struktur hat. Im Falle des Typs `Nullable(Of)` ist das anders, wie das folgende Beispiel zeigt:

Module NullableDemo

```
Sub Main()
    Dim locObj As Object
    Dim locNullOfInt As Nullable(Of Integer) = Nothing

    'Es gibt natürlich eine verwendbare Instanz, denn
    'Nullable(of ) ist ein instanzierter Wertetyp!
    Console.WriteLine("Hat locNullOfInt einen Wert:" & locNullOfInt.HasValue)

    'Und dennoch ergibt das folgende Konstrukt True,
    'als würde locObj keine Referenz haben!
    locObj = locNullOfInt
    Console.WriteLine("Ist locObj Nothing?" & (locObj Is Nothing).ToString)
    Console.WriteLine()

    'Und auch das "Entboxen" geht!
    'Es gibt keine Null-Exception!
    locNullOfInt = DirectCast(locObj, Nullable(Of Integer))

    'Und geht das dann auch? - Natürlich!
    locNullOfInt = DirectCast(Nothing, Nullable(Of Integer))

    'Und noch weiter. Wir boxen einen Nullable(of Integer)
    locNullOfInt = 10
    '
    locObj = locNullOfInt
    Dim locInt As Integer = DirectCast(locObj, Integer)

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

    'Das geht übrigens nicht, obwohl Nullable die Contraints-Einschränkung im
    'Grunde genommen erfüllt!
    'Dim locNullOfInt As Nullable(Of Nullable(Of Integer))
End Sub
End Module
```

Wenn Sie dieses Beispiel ausführen, gibt es die folgenden Zeilen aus:

```
Hat locNullOfInt einen Wert:False  
Ist locObj Nothing?True
```

Taste drücken zum Beenden!

Das, was hier passiert, ist beileibe keine Selbstverständlichkeit – aber dennoch sauberes Design der CLR, denn: Zwar wird `locNullOfInt` nicht initialisiert (oder, um es in diesem Beispiel deutlich zu machen, mit `Nothing` – aber das kommt auf dasselbe raus), aber natürlich existiert dennoch eine Instanz der Struktur. Sie spiegelt eben nur den Wert `Nothing` wider. Gemäß den bekannten Regeln müsste das anschließende Boxen in der Variablen `locObj` auch ergeben, dass `locObj` einen Zeiger auf eine Instanz der `Nothing` widerspiegelnden `locNullOfInt` enthält und keinen Null-Zeiger. Doch das ist nicht der Fall, denn die anschließende Ausgabe von

```
Console.WriteLine("Ist locObj Nothing?" & (locObj Is Nothing).ToString)
```

zeigt

```
Ist locObj Nothing?True
```

auf dem Bildschirm an.

Das »zurückcasten« von `Nothing` in ein `Nullable(Of)` ist damit natürlich genau so gestattet, wie ebenfalls im Listing zu sehen.

Und noch eine Unregelmäßigkeit erfahren Nullables, nämlich wenn es darum geht, einen geboxten Typ (vorausgesetzt er ist eben nicht `Nothing`) in seinen Grundtyp zurückzucasten, wie der folgende Codeausschnitt zeigt:

```
'Und noch weiter. Wir boxen einen Nullable(of Integer)  
locNullOfInt = 10  
'  
locObj = locNullOfInt  
Dim locInt As Integer = DirectCast(locObj, Integer)
```

Hier wird ein `Nullable(Of)`-Datentyp in einem Objekt geboxt, aber später zurück in seinen *Grunddatentypen* gewandelt. Ein, wie ich finde, logisches Design, was allerdings dem »normalen« Vorgehen beim Boxen von Wertetypen in Objekten völlig widerspricht.

Kleine Randnotiz: Dieses Verhalten ist erst zu einem sehr, sehr späten Zeitpunkt beim Entwickeln von Visual Studio 2005 und dem Framework in die CLR eingebaut worden und hat für erhebliche Mehrarbeit bei allen Entwicklerteams und viel zusätzlichen Testaufwand gesorgt. Dass sich Microsoft dennoch für das nunmehr implementierte Verhalten entschieden hat, geht nicht zuletzt auf das Drängen von Kunden und Betatestern zurück, die das Design mit der ursprünglichen, »normalen« CLR-Behandlung von Nullables nicht akzeptieren konnten und als falsch erachteten.

Generische Auflistungen (Generic Collections)

Generische Auflistungen haben im Vergleich zu »herkömmlichen« Auflistungen, die Sie bereits 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 bedeuteten weniger Entwicklungszeit und -kosten.

Nun gibt es nicht wenige generische Auflistungen seit dem 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) Ein Beispiel finden Sie in ► Kapitel 15.	Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung. Besonderheiten: Im Gegensatz zu List(Of type) gibt es überschreibbare Methoden, mit denen man das Verhalten beim Einfügen, Löschen und Neuzuweisen von Elementen der Auflistung in abgeleiteten Klassen beeinflussen kann.
System.Collection.ObjectModel	KeyedCollection(Of key, type) Ein Beispiel finden Sie im ► Abschnitt »KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichem Index-Abfragen« ab Seite 602.	Stellt eine Auflistung von Elementen zur Verfügung, die sowohl das Nachschlagen über einen Schlüssel (Key) eines bestimmten Typs sowie den Einsatz eines Indexers 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 Schlüsseln (Integer, Long, etc.), da es hier beim Serialisieren der Auflistung zu Problemen kommt (Stand: <i>Framework-Version 2.0.50727</i>). ►

Namespace	Auflistung	Beschreibung
System.Collection.ObjectModel	ReadOnlyCollection(Of type)	<p>Stellt eine Auflistung dar, deren Elemente nur gelesen werden können.</p> <p>Besonderheiten: Elemente, die in einer anderen generischen Auflistung gleichen Typs vorliegen, können nur bei der Instanzierung im Konstruktor dieser Auflistung übergeben werden. Ansonsten sind die Elemente nur lesbar und können nach der Instanzierung nicht mehr verändert werden.</p>
System.Collections.Generic	Dictionary(Of key, type)	<p>Stellt eine Auflistung von Schlüsseln und Werten dar.</p> <p>Besonderheiten: Stellt eine Zuordnung von einem Satz von Schlüsseln zu einem Satz von Werten bereit. Jede Hinzufügung zum Wörterbuch besteht aus einem Wert und dem zugeordneten Schlüssel. Ein Wert kann anhand des zugehörigen Schlüssels sehr schnell abgerufen werden (beinahe ein O(1)-Vorgang), da die Dictionary-Klasse in Form einer Hashtable implementiert ist. Mehr zum Konzept von Hashtables finden Sie in ► Kapitel 19.</p>
System.Collections.Generic	LinkedList(Of type) Ein Beispiel finden Sie im ► Abschnitt »Elementverkettungen mit LinkedList(Of)« ab Seite 605.	<p>Stellt eine doppelt verknüpfte Liste dar.</p> <p>Besonderheiten: Ist eine verknüpfte Liste mit einzelnen Knoten vom Typ <code>LinkedListNode</code>; das Einfügen und Entfernen einzelner Elemente geht extrem schnell vonstatten.</p>
System.Collections.Generic	List(Of type) Ein Beispiel finden Sie in ► Kapitel 27.	<p>Stellt eine Standardauflistung für die einfache, unsortierte Verwaltung von Elementen eines bestimmten Typs zur Verfügung.</p> <p>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).</p>
System.Collections.Generic	Queue(Of type)	<p>Stellt eine FIFO-Auflistung (First-In-First-Out) von Objekten dar.</p> <p>Hinweis: Prinzipiell funktioniert diese Auflistung wie ihr nicht generischer Verwandter. Mehr zur nicht generischen Version dieser Auflistung erfahren Sie in ► Kapitel 19.</p>
System.Collections.Generic	SortedDictionary(Of key, type)	<p>Stellt eine Auflistung von Schlüssel-Wert-Paaren dar, deren Reihenfolge anhand des Schlüssels bestimmt wird.</p> <p>Hinweis: Im Gegensatz zu <code>SortedList</code> erfolgt die Sortierung über den Schlüssel und nicht über das verwaltete Element! ►</p>

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 erfahren Sie in ► Kapitel 19. 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 erfahren Sie in ► Kapitel 19.

Tabelle 3.1: Die wichtigsten generischen Auflistungstypen

KeyedCollection – Schlüssel/Wörterbuch-Auflistung mit zusätzlichem Index-Abrufen

Um es ohne Umschweife zu sagen: Die `KeyedCollection` ist in meinen eigenen Projekten neben der `Collection(Of)` am häufigsten anzutreffen. Warum? Sie hat zwei entscheidende Vorteile:

- Sie erlaubt es, einen beliebigen Typ (na ja – *fast* beliebigen Typ, doch dazu später mehr) als Schlüssel anzugeben. Damit können Sie sie als Wörterbuchauflistung verwenden und ihre Elemente beispielsweise über einen Matchcode abfragen.
- Sie können Sie zusätzlich wie eine ganz normale `Collection` behandeln, also mit `For/Each` durch ihre Elemente iterieren oder auch einzelne Elemente über einen numerischen Index abrufen.

Da `KeyedCollection` auch einen Schlüsselwert zum Abrufen jedes ihrer Elemente braucht, benötigt sie einen Mechanismus, der aus dem Element, das es hinzuzufügen gilt, einen Schlüssel erzeugt. Aus diesem Grund ist die `KeyedCollection`-Klasse auch eine abstrakte Basisklasse: Sie müssen die `KeyedCollection` vererben und ihre Methode `GetKeyForItem` überschreiben, in der Sie dann regeln, wie aus dem Element, das es hinzuzufügen gilt, ein eindeutiger Schlüssel kreiert werden soll. Die `KeyedCollection`-Klasse sollte deswegen nur Elemente verwalten, die über wenigstens eine Eigenschaft verfügen, mit der sich ein Element eindeutig von einem anderen unterscheiden lässt. Das kann beispielsweise eine Personalnummer, eine Kundennummer, ein eindeutiger Matchcode oder Ähnliches sein.

WICHTIG: `KeyedCollection` hat aber auch einen entscheidenden Nachteil, oder besser: einen Design-Fehler. Wenn Sie ihren Inhalt serialisieren – also beispielsweise mit einem bestimmten im Framework eingebauten Mechanismus als XML-Datei abspeichern –, können Sie als Schlüssel *keinen* Integer-Datentyp verwenden, da das Framework beim Serialisierungsversuch mit einer Ausnahme »aussteigt«. ► Kapitel 22 geht näher auf dieses Problem ein und hält auch einen Workaround zu dieser »Designunzulänglichkeit« bereit.

BEGLEITDATEIEN: Sie finden das Projekt für das folgende Beispiel im Verzeichnis \VB 2005 - Entwicklerbuch\E - Datentypen\Kap20\KeyedCollectionDemo\.

Im Projekt finden Sie zwei Code-Dateien, *Daten.vb* und *KeyedCollection.vb*. *Daten.vb* enthält eine Adressenklasse, die Sie aus dem vorherigen Kapitel schon kennen. Aufgabe dieser Klasse ist es, die Daten einer Adresse zu speichern. Sie stellt darüber hinaus auch eine statische Funktion bereit, die eine bestimmbare Anzahl von Zufallsadressen generiert und in einer *ArrayList* zurückliefert (mehr zu *ArrayList* finden Sie ebenfalls im vorherigen Kapitel).

Damit die Klasse *Adresse* in diesem Beispiel als Verwaltungselement für die *KeyedCollection* in Frage kommt, muss sie eine Anforderung erfüllen: Sie muss über eine Eigenschaft verfügen, mit der sich ein *Adresse*-Element von jedem anderen *Adresse*-Element unterscheiden lässt.

HINWEIS: Die Funktionalität zur Prüfung auf Eindeutigkeit kann die *Adresse*-Klasse natürlich nicht selbst implementieren, da sie die anderen Elemente einer Auflistung nicht kennt; deswegen ist die Bestimmung einer solchen Dateneigenschaft an dieser Stelle nur eine theoretische Definition. Es ist Aufgabe des späteren Hauptprogramms zu prüfen, ob es keine »Schlüsseldubletten« in einer Elementauflistung gibt, bevor ein neues Element (wie *Adresse*) der Auflistung hinzugefügt wird.

KeyedCollection ist eine *abstrakte* Basisklasse, was bedeutet, dass Sie sie nicht direkt verwenden können. Sie müssen sie vererben und um eine bestimmte Funktionalität erweitern. Deswegen definieren Sie – wie Sie es von abstrakten Klassen gelernt haben – zunächst einen neuen Klassennamen, etwa mit

```
Public Class Adressen
```

und fügen darunter die Zeile

```
Inherits KeyedCollection(Of String, Adresse)
```

ein. Mit dieser Zeile leiten Sie von *KeyedCollection* ab und bestimmen gleichzeitig, dass *String* der Typ für den Schlüssel und *Adresse* der Typ der zu verwaltenden Elemente sein soll.

In dem Moment, in dem Sie die Zeile mit **Eingabe** abschließen, fügt der Visual Basic-Editor automatisch den Rumpf der Methode ein, den Sie in der Basisklasse überschreiben müssen:

```
Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String
```

```
End Function
```

Mit dieser Funktion »holt« sich *KeyedCollection* den Wert, der vom Schlüsseltyp sein muss, immer dann, wenn Sie ein neues Element der Auflistung (beispielsweise durch *Add*) hinzufügen oder ein Element verändern (zum Beispiel mit *Item(Index)=New Adresse(...)*). So wird sichergestellt, dass jedes Element über einen eindeutigen Schlüssel verfügt.

In unserem Beispiel gibt es eine Eigenschaft in der Klasse *Adresse*, die ein Element vom Typ *Adresse* eindeutig kennzeichnet: *Matchcode*. Diese Eigenschaft ist vom Typ *String* – also ist der Schlüsseltyp für die *KeyedCollection* ebenfalls *String*. Die komplett implementierte *KeyedCollection* sieht also folgendermaßen aus:

```

''' <summary>
''' Verwaltet Objekte vom Typ Adresse als Wörterbuch-Auflistung.
''' Abgeleitet von der abstrakten Basisklasse KeyedCollection.
''' </summary>
''' <remarks></remarks>
Public Class Adressen
    Inherits KeyedCollection(Of String, Adresse)

    'Diese Methode muss überschrieben werden, um zu garantieren,
    'dass für jedes Element ein Schlüssel existiert.
    Protected Overrides Function GetKeyForItem(ByVal item As Adresse) As String
        'Hier wird bestimmt, dass der eindeutige Key der Matchcode ist.
        Return item.Matchcode
    End Function
End Class

```

Diese Implementierung ist auch schon alles, was Sie brauchen, um anschließend mit der `KeyedCollection` arbeiten zu können. Das Hauptmodul `KeyedCollection.vb` demonstriert den Umgang und stellt insbesondere heraus, dass Elemente einer auf `KeyedCollection` basierenden Klasse sowohl über einen Index als auch über den entsprechenden Schlüssel abgerufen und bearbeitet werden können:

```

Imports System.Collections.ObjectModel

Module KeyedCollection

    Sub Main()
        'Ersteinmal eine normale ArrayList definieren,
        'die aus 100 Zufallsadressen besteht.
        Dim locArrayList As ArrayList = Adresse.ZufallsAdressen(100)

        'Das ist die "selbst gestrickte" KeyedCollection
        Dim locKeyedAdressen As New Adressen

        'Mit den Elementen der ArrayList füllen wir die KeyedCollection
        For Each locAdressItem As Adresse In locArrayList
            locKeyedAdressen.Add(locAdressItem)
        Next

        'Abrufen der Elemente aus der KeyedCollection:
        'Variation Nr.1 - abrufen über den Index
        For c As Integer = 0 To 10
            Console.WriteLine(locKeyedAdressen(c).ToString)
        Next

        'Leerzeile - der besseren Übersicht wegen:
        Console.WriteLine()

        'Irgendeine herauspicken, um an den Matchcode zu kommen:
        Dim locMatchcode As String = locKeyedAdressen(10).Matchcode
    End Sub
End Module

```

```

'Variation Nr. 2: Eine Adresse kann auch über den Key
'(in diesem Fall über den Matchcode) abgerufen werden.
Dim locAdresse As Adresse = locKeyedAdressen(locMatchcode)
Console.WriteLine("Die Adresse mit dem Matchcode " & locMatchcode & " lautet:")
Console.WriteLine(locAdresse.ToString)
Console.WriteLine()

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

End Sub

End Module

```

Elementverkettungen mit LinkedList(Of)

Erwähnenswert bei den generischen Auflistungen ist die `LinkedList`-Klasse, die das erste Mal mit dem .NET-Framework 2.0 die Möglichkeit zur Verwaltung von Elementen in Form von verketteten Listen einführt.

`LinkedList` stellt eine verknüpfte Liste mit einzelnen Knoten vom Typ `LinkedListNode` dar. Durch das Konzept von `LinkedList`, bei dem die zu speichernden Elemente jeweils selbst auf das nächste und das vorherige Elemente »zeigen«, kann das Einfügen bzw. Löschen von Elementen sehr schnell von-statten gehen, da – sinnbildlich – die Kette an einer Stelle nur aufgeknüpft werden muss, um ein neues Glied in die Kette einzufügen.

Durch die Art der Elementverwaltung bietet `LinkedList` Methoden und Eigenschaften, die Sie in anderen Auflistungen nicht finden. Die folgende Tabelle gibt Aufschluss über die besonderen Methoden und Eigenschaften der `LinkedList`-Klasse.

Elemente werden listenintern nicht wie bei anderen Auflistungsklassen direkt gespeichert, sondern in einem Knoten-Objekt namens `LinkedListNode` abgelegt, das für die Verkettung zum nächsten bzw. vorherigen Knoten der Liste sorgt. Auch diese Klasse ist generisch ausgelegt. Knoten, die der Liste hinzugefügt werden, müssen dabei auf Basis desselben Typs definiert werden, wie die Liste selbst.

Methoden	Beschreibung
<code>AddAfter</code>	Fügt nach einem vorhandenen Knoten in der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddBefore</code>	Fügt vor einem vorhandenen Knoten in der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddFirst</code>	Fügt am Anfang der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>AddLast</code>	Fügt am Ende der <code>LinkedList</code> einen neuen Knoten oder Wert hinzu.
<code>Find</code>	Sucht den ersten Knoten, der den angegebenen Wert enthält.
<code>FindLast</code>	Sucht den letzten Knoten, der den angegebenen Wert enthält.
<code>First</code>	Ruft den ersten Knoten der <code>LinkedList</code> ab. Diese Eigenschaft kann nur gelesen werden.
<code>Last</code>	Ruft den letzten Knoten der <code>LinkedList</code> ab. Diese Eigenschaft kann nur gelesen werden.
<code>Remove</code>	Entfernt das erste Vorkommen eines Knotens oder Werts aus der <code>LinkedList</code> . ▶

Method	Beschreibung
RemoveFirst	Entfernt den Knoten am Anfang der LinkedList.
RemoveLast	Entfernt den Knoten am Ende der LinkedList.

Tabelle 20.2: Die besonderen Methoden und Eigenschaften der LinkedList-Klasse

BEGLEITDATEIEN: Das folgende Beispiel, das den generellen Umgang mit den Klassen `LinkedList` und `LinkedListNode` demonstriert, finden Sie im Verzeichnis `_\\VB 2005 - Entwicklerbuch\\E - Datentypen\\Kap20\\LinkedList\\`.

Module LinkedList

```

Sub Main()
    Dim locLinkedList As New LinkedList(Of Adresse)
    Dim locAdressen As ArrayList = Adresse.ZufallsAdressen(50)
    Dim locAdresse As Adresse = Nothing

    'Acht Adressen an das jeweilige Ende der LinkedList anfügen
    For c As Integer = 0 To 49

        'den 25. merken wir uns für die spätere Suche in der Liste
        If c = 25 Then
            locAdresse = DirectCast(locAdressen(c), Adresse)
        End If
        locLinkedList.AddLast(DirectCast(locAdressen(c), Adresse))
    Next

    Dim locLinkedListNode As LinkedListNode(Of Adresse)
    'Den Knoten finden, der dem 25. Adresseneintrag entsprach.
    locLinkedListNode = locLinkedList.Find(locAdresse)
    Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " & _
        locLinkedListNode.Previous.Value.ToString & _
        " und danach kommt " & locLinkedListNode.Next.Value.ToString)
    Console.WriteLine()

    'Eine neue Adresse vor dem 25. einfügen:
    Console.WriteLine("Sarah Halek vorher einfügen!")
    locLinkedList.AddBefore(locLinkedListNode, _
        New Adresse("SasiMatch", "Halek", "Sarah", "99999", "Musterhausen"))

    Console.WriteLine()

    'Das Gleiche nochmal ausgeben, es sollte die Änderungen jetzt widerspiegeln.
    Console.WriteLine("Vor " & locLinkedListNode.Value.ToString & " kommt " & _
        locLinkedListNode.Previous.Value.ToString & _
        " und danach kommt " & locLinkedListNode.Next.Value.ToString)
    Console.WriteLine()
    Console.WriteLine("Taste drücken zum Beenden")
    Console.ReadKey()
End Sub
End Module

```

Wenn Sie dieses Programm ausführen, gibt es in etwa Folgendes auf dem Bildschirm aus:

Vor "Hörstmann, Hans" kommt "Englisch, Margarete" und danach kommt "Löffelmann, Katrin"

Sarah Halek vorher einfügen!

Vor "Hörstmann, Hans" kommt "Halek, Sarah" und danach kommt "Löffelmann, Katrin"

Taste drücken zum Beenden

Auflistungen und Aktionen (Actions), Aussageprüfer (Predicates) und Vergleiche (Comparisons)

Über das folgende Thema konnte man bislang noch nicht viel erfahren. Auch die Online-Hilfe schweigt sich darüber einigermaßen aus – wie ich finde zu Unrecht, denn ein weiteres Mal zeigt sich, dass man sich mit dem Konzept von generischen Datentypen die Arbeit an einigen Stellen erheblich erleichtern kann.

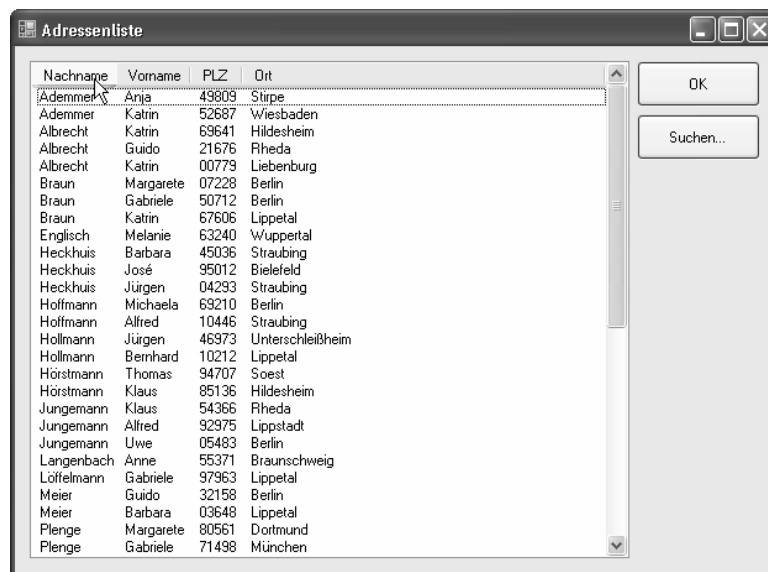


Abbildung 20.2: Sortierung und Suche steuern Sie über die Spaltenköpfe in dieser Adressenliste

BEGLEITDATEIEN: Sie finden das folgende Beispielprojekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\E – Datentypen\Kap20\StaticArrayFunctions\`.

Wenn Sie das Programm starten, sehen Sie einen Dialog, etwa wie Abbildung 20.2 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 Listingausschnitt zu sehen:

```
Sub ElementeDarstellen()

    'Unterdrückt Neuzeichnen-Ereignisse bis zum
    'nächsten EndUpdate; dadurch geht der Aufbau
    'der Elemente schneller und wackelt nicht.
    Me.lvAdressen.BeginUpdate()

    'Alle Elemente der ListView löschen.
    Me.lvAdressen.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.lvAdressen.Columns
        locCol.Width = -2
    Next

    'Aufbau der ListView ist beendet.
    Me.lvAdressen.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
End Sub
```

```

'Zum Wiederfinden Referenz in Tag
locLvwItem.Tag = Element

'Zur Listview hinzufügen
lvwAdressen.Items.Add(locLvwItem)
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. Dieses 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 15) sondern durch die direkte Angabe einer Prozedur mithilfe des `Address Of`-Operators angegeben – sozusagen als Delegatenkonstante. Aber natürlich könnten an dieser Stelle auch »richtige« Delegaten zum Einsatz kommen, die in Abhängigkeit von bestimmten Programmzuständen andere Prozeduren verwendeten, und genau darin besteht der Reiz des Einsatzes dieser Verfahren.

Wichtig dabei ist, dass die Routinen, die Sie mithilfe der `Action`-Klasse aufrufen, den Signaturspruch 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 nehmen, 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 19 gezeigt) sortieren möchten. Den relevanten Codeausschnitt des Beispiels finden Sie im Folgenden:

```

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

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

    'Die Auflistung mithilfe eines Comparison-Delegaten sortieren
    myAdressen.Sort(New Comparison(Of Adresse) (AddressOf AdressenVergleicher))

    'Die Elemente neu sortiert darstellen
    ElementeDarstellen()
End Sub

'Der Comparison-Delegat zum Vergleichen zweier Elemente.
Function AdressenVergleicher(ByVal x As Adresse, ByVal y As Adresse) As Integer

```

```

'Sortierung in Abhängigkeit von zuletzt angeklicktem
'Spaltenkopf durchführen (siehe lvwAdressen_ColumnClick)
Select Case myNach
    Case AdressenSortierenNach.Name
        Return x.Name.CompareTo(y.Name)
    Case AdressenSortierenNach.Ort
        Return x.Ort.CompareTo(y.Ort)
    Case AdressenSortierenNach.PLZ
        Return x.PLZ.CompareTo(y.PLZ)
    Case Else
        Return x.Vorname.CompareTo(y.Vorname)
End Select
End Function

```

Sort arbeitet hier in diesem Beispiel abermals mit einem Delegaten, der durch die generische Comparison-Klasse bereitgestellt wird, und dessen dahinter stehende Methode sich um den eigentlichen Vergleich zweier Objekte (wieder Adresse) kümmert. Damit macht diese Vorgehensweise die Implementierung einer speziellen Comparer-Klasse für die Adresse-Klasse überflüssig.

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 locSuchFormular As New frmSuchen

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

    'Hier kann die Suche beginnen!
    Dim locGefundeneAdresse As Adresse = _
        myAdressen.Find(New Predicate(Of Adresse)(AddressOf AdressenFinder))

    'Wenn ein Element gefunden wurde, dieses markieren.
    If locGefundeneAdresse 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 locGefundeneAdresse 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

```

**'Der Predicate-Delegat zum Suchen eines Elements.
Private Function AdressenFinder(ByVal adr As Adresse) As Boolean**

```

'Gesucht wird immer nach zuletzt ausgewähltem Spaltenkopf.
Select Case myNach
    Case AdressenSortierenNach.Name
        Return adr.Name = myAktuellerSuchbegriff
    Case AdressenSortierenNach.Ort
        Return adr.Ort = myAktuellerSuchbegriff
    Case AdressenSortierenNach.PLZ
        Return adr.PLZ = myAktuellerSuchbegriff
    Case Else
        Return adr.Vorname = myAktuellerSuchbegriff
End Select
End Function

```



Abbildung 20.3: Der Suchbegriff bezieht sich immer auf die Spalte, nach der zuletzt sortiert wurde

In diesem Fall ist etwas mehr Drumherum notwendig, da der Begriff, nach dem gesucht werden soll, zunächst ermittelt werden muss. Und: Der eigentliche Suchbegriff kann der Prozedur, die sich hinter dem der Predicate-Klasse übergebenden Delegaten verbirgt, nicht mitgegeben werden: er muss also der Delegatenprozedur zur Verfügung stehen und wird daher in `myAktuellerSuchbegriff` als Member-Variable gespeichert.

Eine weitere kleine Herausforderung ist das Selektieren des gefundenen Begriffs in der Liste – und das ist im Übrigen auch der Grund, wieso wir 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.

TIPP: Neben `List(Of type)` können Sie auch die generisch ausgelegten, statischen Funktionen von `Array` verwenden, um die hier beschriebenen Arten von Operationen durchzuführen. **Wichtig:** Wenn Sie primitive Datentypen in einer Auflistung speichern, sollten Sie den Einsatz `List(Of type)` anderen Auflistungsklassen vorziehen. 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.
