

Kapitel 5

Arrays und Auflistungen

In diesem Kapitel:

Grundsätzliches zu Arrays	80
Enumeratoren	95
Grundsätzliches zu Auflistungen (Collections)	98
Wichtige Auflistungen der Base Class Library	102

Arrays kennt fast jedes Basic-Derivat seit Jahrzehnten – und natürlich können Sie auch in Visual Basic .NET auf diese Datenfelder (so der deutsche Ausdruck) zurückgreifen. Doch das .NET Framework wäre nicht das .NET Framework, wenn nicht auch Arrays viel mehr Möglichkeiten bieten würden, als nur auf Daten indiziert zuzugreifen.

Das bedeutet, dass die Leistung von Arrays weit über das reine Zur-Verfügung-Stellen von Containern für die Speicherung verschiedener Elemente eines Datentyps hinausreicht. Da Arrays auf Object basieren und damit eine eigene Klasse darstellen (System.Array nämlich), bietet das .NET Framework über Array-Objekte weit-reichende Funktionen zur Verwaltung ihrer Elemente an.

So können Arrays beispielsweise quasi auf Knopfdruck sortiert werden. Liegen sie in sortierter Form vor, können Sie auch binär nach ihren Elementen suchen und viele weitere Dinge mit ihnen anstellen, ohne selbst den entsprechenden Code dafür entwickeln zu müssen.

Zu guter Letzt bildet der Typ System.Array aber auch die Basis für weitere Datentypen – zum Beispiel ArrayList –, die Datenelemente in einer bestimmten Form verwalten können, aber auch die Grundlage für viele der generischen Auflistungstypen, die wiederum die Basis für *LINQ to Objects* bilden.

Dieses Kapitel zeigt Ihnen, was Sie mit Arrays und den von ihnen abgeleiteten Klassen alles anstellen können, und bereitet Sie nicht zuletzt damit auf den Umgang mit *LINQ to Objects* vor.

Grundsätzliches zu Arrays

Arrays im ursprünglichen Basic-Sinne dienen dazu, *mehrere* Elemente desselben Datentyps unter einem bestimmten Namen verfügbar zu machen. Um die einzelnen Elemente zu unterscheiden, bedient man sich eines Indexes (ganz einfach ausgedrückt: einer Nummerierung der Elemente), damit man auf die verschiedenen Array-Elemente zugreifen kann.

HINWEIS Viele der größeren nun folgenden Beispiele sind im Projekt *Arrays* in verschiedenen Methoden zusammengefasst. Das Projekt befindet sich im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\Arrays`. Sie können dieses Projekt verwenden, um die Beispiele an Ihrem eigenen Rechner nachzuvollziehen oder um eigene Experimente mit Arrays durchzuführen.

Ein Beispiel:

```
Sub Beispiel1()  
  
    'Array mit 10 Elementen fest deklarieren.  
    'Wichtig: Anders als in C# oder C++ wird der Index  
    'des letzten Elementes, nicht die Anzahl der Elemente  
    'festgelegt! Elementzählung beginnt bei 0.  
    'Die folgende Anweisung definiert also 10 Elemente:  
    Dim locIntArray(9) As Integer  
  
    'Zufallsgenerator initialisieren,  
    Dim locRandom As New Random(Now.Millisecond)  
  
    For count As Integer = 0 To 9  
        locIntArray(count) = locRandom.Next  
    Next
```

```
For count As Integer = 0 To 9
    Console.WriteLine("Element Nr. {0} hat den Wert {1}", count, locIntArray(count))
Next

Console.ReadLine()

End Sub
```

Wenn Sie dieses Beispiel ausführen, erscheint im Konsolenfenster eine Liste mit Werten, etwa wie im nachstehenden Bildschirmauszug zu sehen (die Werte sollten sich natürlich von Ihren unterscheiden, da es zufällige sind).¹

```
Element Nr. 0 hat den Wert 1074554181
Element Nr. 1 hat den Wert 632329388
Element Nr. 2 hat den Wert 1312197477
Element Nr. 3 hat den Wert 458430355
Element Nr. 4 hat den Wert 1970029554
Element Nr. 5 hat den Wert 503465071
Element Nr. 6 hat den Wert 112607304
Element Nr. 7 hat den Wert 1507772275
Element Nr. 8 hat den Wert 1111627006
Element Nr. 9 hat den Wert 213729371
```

Dieses Beispiel demonstriert die grundsätzliche Anwendung von Arrays. Natürlich sind Sie bei der Definition des Elementtyps nicht auf Integer festgelegt. Es gilt der Grundsatz: Jedes Objekt in .NET kann Element eines Arrays sein.

Änderung der Array-Dimensionen zur Laufzeit

Das Definieren der Array-Größe mit variablen Werten macht Arrays – wie im Beispiel des letzten Abschnittes gezeigt – zu einem sehr flexiblen Werkzeug bei der Verarbeitung von großen Datenmengen. Doch Arrays sind noch flexibler: Mit der `ReDim`-Anweisung gibt Ihnen Visual Basic die Möglichkeit, ein Array noch nach seiner ersten Deklaration neu zu dimensionieren. Damit werden selbst einfache Arrays zu dynamischen Datencontainern. Auch hier soll ein Beispiel den Umgang verdeutlichen:

```
Sub Beispiel13()

    Dim locAnzahlStrings As Integer = 15
    Dim locStringArray As String()
    locStringArray = GeneriereStrings(locAnzahlStrings, 30)
    Console.WriteLine("Ausgangsgröße: {0} Elemente. Es folgt der Inhalt:", locStringArray.Length)
    Console.WriteLine(New String("c", 40))
    DruckeStrings(locStringArray)

End Sub
```

¹ Kleine Anmerkung am Rande: Ganz so zufällig sind die Werte nicht – Random stellt nur sicher, dass generierte Zahlenfolgen zufällig *verteilt* sind. Bei gleicher Ausgangsbasis (definiert durch den Parameter `Seed`, den Sie der `Random`-Klasse beim Instanzieren übergeben) produziert `Random` auch gleiche Zahlenfolgen. Da wir hier die Millisekunde als Basis übergeben, und eine Sekunde aus 1000 Millisekunden besteht, gibt es eine Wahrscheinlichkeit von 1:1000, dass Sie dieselbe wie die hier abgedruckte Zahlenfolge generieren lassen.

```

'Wir brauchen 10 weitere, die alten sollen aber erhalten bleiben!
ReDim Preserve locStringArray(locStringArray.Length + 9)

'Bleiben die alten wirklich erhalten?
Console.WriteLine()
Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
Console.WriteLine(New String("c", 40))
DruckeStrings(locStringArray)

'10 weitere Elemente generieren.
Dim locTempStrings(9) As String

'10 Zeichen mehr pro Element, sodass wir die neuen leicht erkennen können.
locTempStrings = GeneriereStrings(10, 40)

'In das "alte" Array kopieren, aber ab Index 15,
locTempStrings.CopyTo(locStringArray, 15)

'und nachschauen, was nun wirklich drinsteht!
Console.WriteLine()
Console.WriteLine("Inhaltsüberprüfung:", locStringArray.Length)
Console.WriteLine(New String("c", 40))
DruckeStrings(locStringArray)

Console.ReadLine()
End Sub

```

Dieses Beispiel macht sich die Möglichkeit zunutze (direkt in der ersten Codezeile), die Dimensionierung und Deklaration eines Arrays zeitlich voneinander trennen zu können. Das Array `locStringArray` wird zunächst nur als Array deklariert – wie groß es sein soll, wird zu diesem Zeitpunkt noch nicht bestimmt. Dabei spielt es in Visual Basic übrigens keine Rolle, ob Sie eine Variable in diesem

```
Dim locStringArray As String()
```

oder diesem

```
Dim locStringArray() As String
```

Stil als Array deklarieren.

Die Größe des Arrays wird das erste Mal von der Prozedur `GeneriereString` festgelegt. Hier erfolgt zwar die Dimensionierung eines zunächst völlig anderen Arrays (`locStrings`); da dieses Array aber als Rückgabewert der aufrufenden Instanz überlassen wird, lebt der hier erstellte Array-Inhalt unter anderem Namen (`locStringArray`) weiter. Das durch beide Objektvariablen referenzierte Array ist dasselbe (im ursprünglichen Sinne des Wortes).

Übrigens: Diese Vorgehensweise entspricht eigentlich schon einem typischeren Neudimensionieren eines Arrays. Wie Sie beim ersten Array-Beispiel gesehen haben, spielt es natürlich keine Rolle, ob Sie eine Array-Variablen zur Aufnahme eines Array-Rückgabeparameters verwenden, die zuvor mit einer festen Anzahl an Elementen oder ohne die Angabe der Array-Größe dimensioniert wurde. Allerdings: Sie verlieren bei dieser Vorgehensweise den Inhalt des ursprünglichen Arrays, denn die Unterroutine erstellt ein neues Array, und mit der Zuweisung an die Objektvariable `locStringArray` wird intern nur ein Zeiger umgebogen. Der Speicherbereich der alten Elemente ist nicht mehr referenzierbar.

WICHTIG Diese Tatsache hat Konsequenzen, denn: Anders, als es bei Visual Basic 6.0 noch der Fall war, werden Arrays bei einer Zuweisung an eine andere Objektvariable *nicht* automatisch kopiert. Array-Variablen verhalten sich so wie jeder andere Referenztyp auch unter .NET: Ein Zuweisen einer Array-Variablen an eine andere biegt nur ihren Zeiger auf den Speicherbereich der eigentlichen Daten im Managed Heap um. Die Elemente bleiben an ihrem Platz im Managed Heap, und es wird kein neuer Speicherbereich mit einer Kopie der Array-Elemente für die neue Objektvariable erzeugt!

Das Redimensionieren kann nicht nur über Zuweisungen, sondern – wie im Beispielcode zu sehen – auch mit der `ReDim`-Anweisung erfolgen. Mit dem zusätzlichen Schlüsselwort `Preserve` haben Sie darüber hinaus die Möglichkeit zu bestimmen, dass die alten Elemente dabei erhalten bleiben. Man möchte meinen, dass diese Verhaltensweise die Regel sein sollte, doch mit dem Wissen im Hinterkopf, was beim Neudimensionieren mit `ReDim` genau passiert, sieht die Sache schon anders aus:

- Wird `ReDim` aufgerufen, wird ein komplett neuer Speicherbereich dafür reserviert.
- Der Zeiger für die Objektvariable auf den Bereich für die zuvor zugeordneten Array-Elemente wird auf den neuen Speicherbereich umgebogen.
- Der Speicherbereich, der die alten Array-Elemente enthielt, fällt dem nächsten Garbage-Collector-Durchlauf zum Opfer.
- Wenn `Preserve` hinter der `ReDim`-Anweisung platziert wird, bleiben die alten Array-Elemente erhalten. Doch das entspricht nicht der vollständigen Erklärung des Vorgangs. In Wirklichkeit wird auch hier ein neuer Speicherbereich erstellt, der den Platz für die neu angegebene Anzahl an Array-Elementen bereithält. Auch bei `Preserve` wird der Zeiger auf den Speicherbereich mit den alten Elementen für die betroffene Objektvariable auf den neuen Speicherbereich umgebogen. Doch bevor der alte Speicherbereich freigegeben wird und sich der Garbage Collector über die alten Elemente hermachen kann, werden die vorhandenen Elemente in den neuen Bereich kopiert.

Aus diesem Grund können Sie mit `Preserve` nur Elemente retten, die in eindimensionalen Arrays gespeichert sind oder die durch die letzte Dimension eines mehrdimensionalen Arrays angesprochen werden.

Wertevorbelegung von Array-Elementen im Code

Alle Arrays, die in den vorangegangenen Beispielen verwendet wurden, sind zur Laufzeit mit Daten gefüllt worden. In vielen Fällen möchten Sie aber Arrays erstellen, die Sie automatisch mit Daten vorbelegen, die das Programm fest vorgibt:

```
Sub Beispiel5()  
  
    'Deklaration und Definition von Elementen mit Double-Werten  
    Dim locDoubleArray As Double() = {123.45F, 5435.45F, 3.14159274F}  
  
    'Deklaration und spätere Definition von Elementen mit Integer-Werten  
    Dim locIntegerArray As Integer()  
    locIntegerArray = New Integer() {1I, 2I, 3I, 3I, 4I}  
    'Deklaration und spätere Definition von Elementen mit Date-Werten  
    Dim locDateArray As Date()  
    locDateArray = New Date() {#12/24/2005#, #12/31/2005#, #3/31/2006#}
```

```
'Deklaration und Definition von Elementen im Char-Array:
Dim locCharArray As Char() = {"V"c, "B"c, "."c, "N"c, "E"c, "T"c, " "c, _
    "r"c, "u"c, "l"c, "e"c, "s"c, "!c}

'Zweidimensionales Array
Dim locZweiDimensional As Integer(,)
locZweiDimensional = New Integer(,) {{10, 10}, {20, 20}, {30, 30}}

'Oder verschachtelt (das ist nicht Zwei-Dimensional)!
Dim locVerschachtelt As Date()()
locVerschachtelt = New Date()() {New Date() {#12/24/2004#, #12/31/2004#}, _
    New Date() {#12/24/2005#, #12/31/2005#}}

End Sub
```

Häufigste Fehlerquelle bei dieser Vorgehensweise: Der Zuweisungsoperator wird falsch gesetzt. Bei der kombinierten Deklaration/Definition wird er benötigt; *definieren* Sie nur neu, lassen Sie ihn weg. Beachten Sie auch den Unterschied zwischen mehrdimensionalen und verschachtelten Arrays, auf den ich im nächsten Abschnitt genauer eingehen möchte.

Mehrdimensionale und verschachtelte Arrays

Bei der Definition von Arrays sind Sie nicht auf eine Dimension beschränkt – das ist ein Feature, das schon bei jahrzehntealten Basic-Dialekten zu finden ist. Sie können ein mehrdimensionales Array erstellen, indem Sie bei der Deklaration die Anzahl der Elemente für jede Dimension durch Komma getrennt angeben:

```
Dim DreiDimensional(5, 10, 3) As Integer
```

Möchten Sie die Anzahl der zu verwaltenden Elemente bei der Deklaration des Arrays nicht festlegen, verwenden Sie die folgende Deklarationsanweisung:

```
Dim AuchDreiDimensional As Integer(,,)
```

Mit `ReDim` oder dem Aufruf von Funktionen, die ein entsprechend dimensioniertes Array zurückliefern, können Sie anschließend das Array definieren.

Verschachtelte Arrays

Verschachtelte Arrays sind etwas anders konzipiert als mehrdimensionale Arrays. Bei verschachtelten Arrays ist ein Array-Element selbst ein Array (welches auch wieder Arrays beinhalten kann usw.). Anders als bei mehrdimensionalen Arrays können die einzelnen Elemente unterschiedlich dimensionierte Arrays enthalten, und diese Zuordnung lässt sich auch im Nachhinein noch ändern.

Verschachtelte Arrays definieren Sie, indem Sie die Klammerpaare mit der entsprechenden Array-Dimension hintereinander schreiben – anders als bei mehrdimensionalen Arrays, bei denen die Dimensionen in einem Klammerpaar mit Komma getrennt angegeben werden.

Die folgenden Beispiel-Codezeilen (aus der Sub `Beispiel6`) zeigen, wie Sie verschachtelte Arrays definieren, deklarieren und ihre einzelnen Elemente abrufen können:

```
'Einfach verschachtelt; Tiefe wird nicht definiert.
Dim EinfachVerschachtelt(10)() As Integer

'Erstes Element hält ein Integer-Array mit drei Elementen.
EinfachVerschachtelt(0) = New Integer() {10, 20, 30}

'Zweites Element hält ein Integer-Array mit acht Elementen.
EinfachVerschachtelt(1) = New Integer() {10, 20, 30, 40, 50, 60, 70, 80}

'Druckt das dritte Element des zweiten Elementes (30) des Arrays.
Console.WriteLine(EinfachVerschachtelt(1)(2))

'In einem Rutsch alles neu zuweisen.
EinfachVerschachtelt = New Integer()() {New Integer() {30, 20, 10},
                                         New Integer() {80, 70, 60, 50, 40, 30, 20, 10}}

'Druckt das dritte Element des zweiten Elementes (jetzt 60) des Arrays.
Console.WriteLine(EinfachVerschachtelt(1)(2))
Console.ReadLine()
```

Die wichtigsten Eigenschaften und Methoden von Arrays

In den vorangegangenen Beispielprogrammen ließ es sich nicht vermeiden, die eine oder andere Eigenschaft oder Methode des Array-Objektes bereits anzuwenden. Dieser Abschnitt soll sich ein wenig genauer mit den zusätzlichen Möglichkeiten dieses Objektes beschäftigen – denn sie sind mächtig und können Ihnen, richtig angewendet, eine Menge Entwicklungszeit ersparen.

Anzahl der Elemente eines Arrays ermitteln mit Length

Wenn Sie wissen wollen, wie viele Elemente ein Array beherbergt, bedienen Sie sich seiner Length-Eigenschaft. Bei zwei- und mehrdimensionalen Arrays ermittelt Length ebenfalls die Anzahl aller Elemente.

Aufgepasst bei verschachtelten Arrays: Hier ermittelt Length nämlich nur die Elementanzahl des umgebenden Arrays. Sie können die Array-Länge eines Elementes des umgebenden Arrays etwa so ermitteln:

```
'Verschachtelte Arrays
Dim locVerschachtelt As Date()()
locVerschachtelt = New Date()() {New Date() {#12/24/2004#, #12/31/2004#}, _
                                New Date() {#12/24/2005#, #12/31/2005#}}

Console.WriteLine("Äußeres Array hat {0} Elemente.", locVerschachtelt.Length)
Console.WriteLine("Array des 1. Elements hat {0} Elemente.", locVerschachtelt(0).Length)
```

Sortieren von Arrays mit Array.Sort

Arrays lassen sich durch eine ganz simple Methode sortieren. Das folgende Beispiel soll demonstrieren, wie es geht:

```
Sub Beispiel7()

    'Array-Erstellen:
    Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _
                                "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _
```

```

        "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrin", "Momo", _
        "Gareth", "Anne"}
    System.Array.Sort(1ocNamen)
    DruckeStrings(1ocNamen)
    Console.ReadLine()
End Sub

```

Wenn Sie dieses Beispiel starten, produziert es folgendes Ergebnis im Konsolenfenster:

```

Andrea
Anja
Anne
Daja
Flori
Gaby
Gareth
Hanna
José
Jürgen
Katrin
Klaus
Kricke
Leonie-Gundula
Martina
Melanie
Michaela
Miriam
Momo
Myriam
Thomas
Ute
Uwe

```

Die Sort-Methode ist eine statische Methode von System.Array und sie kann noch eine ganze Menge mehr. So haben Sie beispielsweise die Möglichkeit, einen korrelierenden Index mitsortieren zu lassen, oder Sie können bestimmen, zwischen welchen Indizes eines Arrays die Sortierung stattfinden soll. Die Online-Hilfe zum .NET Framework verrät Ihnen, welche Überladungen die Sort-Methode genau anbietet.

Umdrehen der Array-Anordnung mit Array.Reverse

Wichtig in diesem Zusammenhang ist eine weitere statische Methode von System.Array namens Reverse, die die Reihenfolge der einzelnen Array-Elemente umkehrt. Im Zusammenhang mit der Sort-Methode erreichen Sie durch den anschließenden Einsatz von Reverse die Sortierung eines Arrays in absteigender Reihenfolge. Wenn Sie das vorherige Beispiel um diese Zeilen

```

    Console.WriteLine()
    Console.WriteLine("Absteigend sortiert:")
    Array.Reverse(1ocNamen)
    DruckeStrings(1ocNamen)
    Console.ReadLine()

```

ergänzen, sehen Sie schließlich die Namen in umgekehrter Reihenfolge im Konsolenfenster, etwa:

```
Absteigend sortiert:  
Uwe  
Ute  
Thomas  
.br/>.br/>.br/>Anja  
Andrea
```

Durchsuchen eines sortierten Arrays mit `Array.BinarySearch`

Auch bei der Suche nach bestimmten Elementen eines Arrays ist Ihnen das .NET Framework behilflich. Dazu stellt die `Array`-Klasse die statische Funktion `BinarySearch` zur Verfügung.

WICHTIG Damit eine binäre Suche in einem Array durchgeführt werden kann, muss das Array in sortierter Form vorliegen – ansonsten wird die Funktion höchstwahrscheinlich falsche Ergebnisse zurückliefern. Wirklich brauchbar ist die Funktion überdies nur dann, wenn Sie sicherstellen, dass es keine Dubletten in den Elementen gibt. Da eine binäre Suche erfolgt, ist nicht gewährleistet, ob die Funktion das erste zutreffende Objekt findet oder ein beliebiges, das dem Gesuchten entspricht.

Beispiel:

```
Sub Beispiel8()  
  
    'Array-Erstellen:  
    Dim locNamen As String() = {"Jürgen", "Martina", "Hanna", "Gaby", "Michaela", "Miriam", "Ute", _  
                                "Leonie-Gundula", "Melanie", "Uwe", "Andrea", "Klaus", "Anja", _  
                                "Myriam", "Daja", "Thomas", "José", "Kricke", "Flori", "Katrin", "Momo", _  
                                "Gareth", "Anne", "Jürgen", "Gaby"}  
  
    System.Array.Sort(locNamen)  
    Console.WriteLine("Jürgen wurde gefunden an Position {0}", _  
                      System.Array.BinarySearch(locNamen, "Jürgen"))  
    Console.ReadLine()  
End Sub
```

Wie `Sort` ist auch `BinarySearch` eine überladene Funktion und bietet weitere Optionen, die die Suche beispielsweise auf bestimmte Indexpbereiche beschränkt. IntelliSense und die Online-Hilfe geben hier genauere Hinweise für die Verwendung.

Implementierung von `Sort` und `BinarySearch` für eigene Klassen

Das .NET Framework erlaubt es, Arrays von beliebigen Typen zu erstellen, auch von solchen, die Sie selbst erstellt haben. Bei der Erstellung einer Klasse, die als Element eines Arrays fungieren soll, brauchen Sie dabei nichts Besonderes zu beachten.

Anders wird das allerdings, wenn Sie eine Funktion auf das Array anwenden wollen, die einen Elementvergleich erfordert, oder wenn Sie sogar steuern wollen, nach welchen Kriterien Ihr Array beispielsweise sortiert werden soll oder auch nach welchem Kriterium Sie es mit `BinarySearch` durchsuchen lassen möchten. Dazu ein Beispiel:

Sie haben eine Klasse entwickelt, die die Adressen einer Kontaktdatenbank speichert. Der nachfolgend gezeigte Code demonstriert, wie sie funktioniert und wie man sie anwendet:

BEGLEITDATEIEN

Im Folgenden sehen Sie zunächst den Klassencode, der einen Adresseneintrag verwaltet – dieses Projekt finden Sie im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\IComparer01`

```
Public Class Adresse

    Protected myName As String
    Protected myVorname As String
    Protected myPLZ As String
    Protected myOrt As String

    Sub New(ByVal Name As String, ByVal Vorname As String, ByVal Plz As String, ByVal Ort As String)
        myName = Name
        myVorname = Vorname
        myPLZ = Plz
        myOrt = Ort
    End Sub

    Public Property Name() As String
        Get
            Return myName
        End Get
        Set(ByVal Value As String)
            myName = Value
        End Set
    End Property

    Public Property Vorname() As String
        Get
            Return myVorname
        End Get
        Set(ByVal Value As String)
            myVorname = Value
        End Set
    End Property

    Public Property PLZ() As String
        Get
            Return myPLZ
        End Get
        Set(ByVal Value As String)
            myPLZ = Value
        End Set
    End Property

    Public Property Ort() As String
        Get
            Return myOrt
        End Get
        Set(ByVal Value As String)
            myOrt = Value
        End Set
    End Property

End Class
```

```

Public Overrides Function ToString() As String
    Return Name + ", " + Vorname + ", " + PLZ + " " + Ort
End Function
End Class

```

Sie sehen selbst: einfachstes Visual Basic. Die Klasse stellt ein paar Eigenschaften für die von ihr verwalteten Daten zur Verfügung, und sie überschreibt die ToString-Methode der Basis-Klasse, damit sie eine Adresse als kompletten String ausgeben kann.²

Das folgende kleine Beispielprogramm definiert ein Array aus dieser Klasse, richtet ein paar Adressen zum Experimentieren ein und druckt diese anschließend in einer eigenen Unterroutine aus:

```

Module ComparerBeispiel

    Sub Main()
        Dim locAdressen(5) As Adresse

        locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
        locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
        locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
        locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
        locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
        locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

        Console.WriteLine("Adressenliste:")
        Console.WriteLine(New String("=", 40))
        DruckeAdressen(locAdressen)
        'Array.Sort(locAdressen)
        Console.ReadLine()
    End Sub

    Sub DruckeAdressen(ByVal Adressen As Adresse())
        For Each locString As Adresse In Adressen
            Console.WriteLine(locString)
        Next
    End Sub

End Module

```

Auch hier werden Sie erkennen: Es passiert nichts wirklich Aufregendes. Wenn Sie das Programm starten, produziert es, wie zu erwarten, folgende Ausgabe im Konsolenfenster:

```

Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund

```

² Überschreiben bedeutet, dass ein Element der Basisklasse, von der aus eine Klasse abgeleitet wurde, durch ein neues ersetzt wird. Da jede Klasse, die Sie neu erstellen, implizit von Object erbt, verfügt jede Klasse auch automatisch über die Funktionen, die Object kennt – die ToString-Methode, die eine Objektbeschreibung als Zeichenkette ermittelt, gehört dazu. Um eine Methode zu implementieren, die den Inhalt eines Objektes als String ausgibt, ergibt es in jedem Fall Sinn, dazu die immer vorhandene ToString-Methode durch eine neue zu ersetzen und diese dann weiter zu verwenden. Mehr zum Thema Vererbung, Polymorphie und Überschreiben von Methoden finden Sie im Buch *Visual Basic 2005 – das Entwicklerbuch*, das sie unter www.activedevelop.de kostenlos herunterladen können.

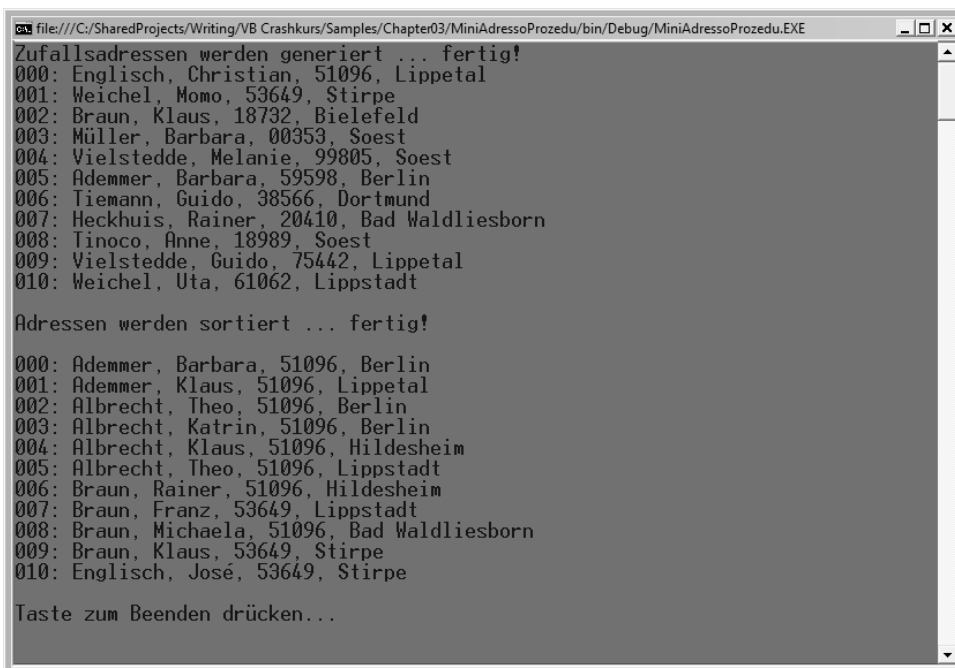
```
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten
```

Die Liste, wie wir Sie anzeigen lassen, ist allerdings ein ziemliches Durcheinander. Beobachten Sie, was passiert, wenn wir das Programm um eine Sort-Anweisung ergänzen und wie die Liste anschließend aussieht. Dazu nehmen Sie die Auskommentierung der Sort-Methode im Listing einfach zurück.

Nach dem Start des Programms warten Sie vergeblich auf die zweite, sortierte Ausgabe der Liste. Stattdessen löst das .NET Framework eine Ausnahme aus, etwa wie in Abbildung 5.1 zu sehen.

Der Grund dafür: Die Sort-Methode der Array-Klasse versucht, die einzelnen Elemente des Arrays miteinander zu vergleichen. Dazu benötigt es einen so genannten *Comparer* (etwa: *Vergleicher*). Da wir nicht explizit angeben, dass wir einen speziellen Comparer verwenden wollen (Welchen auch? – Wir haben noch keinen!), erzeugt es einen Standard-Comparer, der aber wiederum die Einbindung einer bestimmten Schnittstelle in der Klasse verlangt, deren Elemente er miteinander vergleichen soll. Diese Schnittstelle nennt sich *IComparable*. Leider haben wir auch diese Schnittstelle nicht implementiert, und die Ausnahme ist die Folge.

Wir haben nun drei Möglichkeiten. Wir binden die *IComparable*-Schnittstelle ein, dann können Elemente unserer Klasse auch ohne die Nennung eines expliziten Comparer verglichen und im Endeffekt sortiert werden.



```
file:///C:/SharedProjects/Writing/VB Crashkurs/Samples/Chapter03/MiniAdressoProzedu/bin/Debug/MiniAdressoProzedu.EXE
Zufallsadressen werden generiert ... fertig!
000: Englisch, Christian, 51096, Lippetal
001: Weichel, Momo, 53649, Stirpe
002: Braun, Klaus, 18732, Bielefeld
003: Müller, Barbara, 00353, Soest
004: Vielstedde, Melanie, 99805, Soest
005: Ademmer, Barbara, 59598, Berlin
006: Tiemann, Guido, 38566, Dortmund
007: Heckhuis, Rainer, 20410, Bad Waldliesborn
008: Tinoco, Anne, 18989, Soest
009: Vielstedde, Guido, 75442, Lippetal
010: Weichel, Uta, 61062, Lippstadt

Adressen werden sortiert ... fertig!
000: Ademmer, Barbara, 51096, Berlin
001: Ademmer, Klaus, 51096, Lippetal
002: Albrecht, Theo, 51096, Berlin
003: Albrecht, Katrin, 51096, Berlin
004: Albrecht, Klaus, 51096, Hildesheim
005: Albrecht, Theo, 51096, Lippstadt
006: Braun, Rainer, 51096, Hildesheim
007: Braun, Franz, 53649, Lippstadt
008: Braun, Michaela, 51096, Bad Waldliesborn
009: Braun, Klaus, 53649, Stirpe
010: Englisch, José, 53649, Stirpe

Taste zum Beenden drücken...
```

Abbildung 5.1 Wenn Sie das Programm starten, löst es eine Ausnahme aus, sobald die Sort-Methode der Array-Klasse erreicht ist – in der Detailbeschreibung zur Ausnahme sehen Sie, was falsch läuft!

Oder wir stellen der Klasse einen *expliziten* Comparer zur Verfügung; in diesem Fall müssen wir ihn beim Einsatz von Sort (oder auch BinarySearch) benennen. Dieser Comparer hätte den Vorteil, dass sich durch ihn steuern ließe, nach welchem Kriterium die Klasse durchsucht bzw. sortiert werden soll.

Die dritte Möglichkeit: Wir machen beides. Damit wird unsere Klasse universell nutzbar und läuft nicht Gefahr, eine Ausnahme auslösen zu können. Und genau das werden wir in den nächsten beiden Abschnitten in die Tat umsetzen.

Implementieren der Vergleichsfähigkeit einer Klasse durch IComparable

Die Implementierung der IComparable-Schnittstelle, damit Instanzen unserer Klasse miteinander verglichen werden können, ist vergleichsweise simpel.

Erinnern wir uns: Damit bestimmter Code allgemeingültig verwendbar sein kann, kann man sich des Konzeptes von Schnittstellen (engl. *Interfaces*) bedienen. Das Einbinden einer Schnittstelle in eine Klasse oder eine Struktur erzwingt, dass in der Klasse die Elemente (Eigenschaften, Methoden, Ereignisse) vorhanden sein müssen, die die Schnittstelle vorschreibt. Gleichzeitig reicht es aus, beispielsweise eine Methode, die eine Schnittstelle für eine Klasse vorschreibt, über eine Schnittstellenvariable aufzurufen. Damit spielt es für die aufrufende Instanz keine Rolle mehr, mit was sie es genau zu tun hat – ob es beispielsweise ein Integer oder ein Decimal-Datentyp ist. Die aufrufende Instanz arbeitet lediglich mit einer Schnittstellenvariablen, und ruft über diese die gewünschte Methode auf. Bei einem Integer-Datentyp, der diese Schnittstelle einbindet, werden so beispielsweise Integer-Werte verglichen, bei einem String-Datentyp, der die gleiche Schnittstelle einbindet, eben Zeichenketten. Doch das merkt und interessiert die aufrufende Instanz gar nicht (in diesem Fall die Array-Klasse des .NET Frameworks). Sie ist nur am Ergebnis interessiert.

Das Implementieren der Schnittstelle in unserer Beispielklasse erfordert übrigens zusätzlich lediglich das Vorhandensein einer CompareTo-Methode, die die aktuelle Instanz der Klasse mit einer weiteren vergleicht, und das Anzeigen, dass es sich bei der Methode um die von der Schnittstelle vorgeschriebene Methode handelt.

Da durch den Einsatz von IComparable keine Möglichkeit besteht festzulegen, welches Datenfeld das Vergleichskriterium sein soll, wird unser Kriterium eine Zeichenkette sein, die aus Namen, Vornamen, Postleitzahl und Ort besteht – genau die Zeichenkette also, die ToString in der jetzigen Version bereits zurückliefert. Deswegen brauchen wir den eigentlichen Vergleich noch nicht einmal selbst durchzuführen, sondern können ihn an die CompareTo-Funktion des String-Objektes, das wir von ToString zurückerhalten, weiterreichen. Die Modifizierungen an der Klasse sind also denkbar gering (aus Platzgründen nur gekürzter Code, der die Änderungen widerspiegelt):

```
Public Class Adresse
    Implements IComparable
    .
    .
    .
    Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo

        Dim locAdresse As Adresse

        Try
            locAdresse = DirectCast(obj, Adresse)
        Catch ex As InvalidCastException
            Dim up As New InvalidCastException("'CompareTo' der Klasse 'Adresse' kann keine Vergleiche " + _
                "mit Objekten anderen Typs durchführen!")

            Throw up
        End Try
        Return ToString.CompareTo(locAdresse.ToString)
    End Function
End Class
```

An dieser Stelle vielleicht erwähnenswert ist der fett gekennzeichnete mittlere Bereich im Programm. Auf den ersten Blick mag es unsinnig erscheinen, einen möglichen Fehler abzufangen und ihn anschließend mehr oder weniger unverändert wieder auszulösen. Aber: Der Fehlertext ist entscheidend, und Sie tun sich selbst einen Gefallen, wenn Sie den Fehlertext einer Ausnahme, die Sie generieren, so formulieren, dass Sie eindeutig wissen, wer oder was sie ausgelöst hat. Wenn Sie das Programm anschließend starten, liefert es das gewünschte Ergebnis, wie im Folgenden zu sehen:

```
Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten

Adressenliste (sortiert):
=====
Ademmer, Uta, 55555 Bad Waldholz
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Kaiser, Wilhelm, 12121 Ostenwesten
Löffelmann, Klaus, 11111 Soest
Sonntag, Christian, 33333 Wuppertal
Sonntag, Miriam, 22222 Dortmund
```

Implementieren einer gesteuerten Vergleichsfähigkeit durch IComparer

So weit, so gut – unser Beispielprogramm läuft immerhin schon, und die Elemente des Arrays lassen sich sortieren. Aber: Das Programm sortiert stumpf nach dem Nachnamen – und diese Verhaltensweise können wir ihm ohne weitere Maßnahmen auch nicht abgewöhnen.

Was die Adresse-Klasse braucht, ist eine Art Steuerungseinheit, die durch Setzen bestimmter Eigenschaften festlegt, wie Vergleiche stattfinden sollen. Wenn Sie sich zum Beispiel die Sort-Funktion von System.Array ein wenig genauer anschauen, werden Sie feststellen, dass sie als optionalen Parameter eine Variable vom Typ IComparer entgegennimmt.

Eine Klasse, die IComparer einbindet, macht genau das Verlangte. Sie kann den Vergleichsvorgang beeinflussen. Wenn Sie IComparer in einer Klasse implementieren, müssen Sie auch die Funktion Compare in dieser Klasse zur Verfügung stellen. Dieser Funktion werden zwei Objekte übergeben, die die Funktion miteinander vergleichen soll. Ist eines der Objekte größer (was auch immer das heißt, denn es ist bis dahin ein abstraktes Attribut), liefert sie den Wert 1, ist es kleiner, den Wert -1, und ist es gleich, liefert sie den Wert 0 zurück.

Ein Comparer steht in keinem direkten Verhältnis zu den Klassen, die er vergleichen soll; er wird also nicht durch weitere Schnittstellen reglementiert. Aus diesem Grund muss der Comparer selber dafür Sorge tragen, dass ihm nur die Objekte zum Vergleichen angeliefert werden, die er verarbeiten will. Bekommt er andere, muss er eine Ausnahme auslösen.

In unserem Fall muss der Comparer noch ein bisschen mehr können. Er muss die Funktionalität zur Verfügung stellen, durch die der Entwickler steuern kann, nach *welchem* Kriterium der Adresse-Klasse er vergleichen will. Aus diesen Gründen ergibt sich folgender Code für einen Comparer unseres Beispiels:

BEGLEITDATEIENSie finden das Projekt im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\IComparer02`

```
'Nur für den einfachen Umgang mit der Klasse.
Public Enum ZuVergleichen
    Name
    PLZ
    Ort
End Enum

Public Class AdressenVergleicher
    Implements IComparer

    'Speichert die Einstellung, nach welchem Kriterium verglichen wird.
    Protected myZuVergleichenMit As ZuVergleichen

    Sub New(ByVal ZuVergleichenMit As ZuVergleichen)
        myZuVergleichenMit = ZuVergleichenMit
    End Sub

    Public Function Compare(ByVal x As Object, ByVal y As Object) As Integer _
        Implements System.Collections.IComparer.Compare
        'Nur erlaubte Typen durchlassen:
        If (Not (TypeOf x Is Adresse)) Or (Not (TypeOf y Is Adresse)) Then
            Dim up As New InvalidCastException("'Compare' der Klasse 'AdressenVergleicher' kann nur " +
                "Vergleiche vom Typ 'Adresse' durchführen!")
            Throw up
        End If

        'Beide Objekte in den richtigen Typ casten, damit das Handling einfacher wird:
        Dim locAdr1 As Adresse = DirectCast(x, Adresse)
        Dim locAdr2 As Adresse = DirectCast(y, Adresse)

        'Hier passiert die eigentliche Steuerung,
        'nach welchem Kriterium verglichen wird:
        If myZuVergleichenMit = ZuVergleichen.Name Then
            Return locAdr1.Name.CompareTo(locAdr2.Name)
        ElseIf myZuVergleichenMit = ZuVergleichen.Ort Then
            Return locAdr1.Ort.CompareTo(locAdr2.Ort)
        Else
            Return locAdr1.PLZ.CompareTo(locAdr2.PLZ)
        End If

    End Function

    'Legt die Vergleichseinstellung offen.
    Public Property ZuVergleichenMit() As ZuVergleichen
        Get
            Return myZuVergleichenMit
        End Get
        Set(ByVal Value As ZuVergleichen)
            myZuVergleichenMit = Value
        End Set
    End Property
End Class
```

Zum Beweis, dass alles wie gewünscht läuft, ergänzen Sie das Hauptprogramm um folgende Zeilen (fettgedruckt im folgenden Listing):

```
Module ComparerBeispiel

Sub Main()
    Dim locAdressen(5) As Adresse

    locAdressen(0) = New Adresse("Löffelmann", "Klaus", "11111", "Soest")
    locAdressen(1) = New Adresse("Heckhuis", "Jürgen", "99999", "Gut Uhlenbusch")
    locAdressen(2) = New Adresse("Sonntag", "Miriam", "22222", "Dortmund")
    locAdressen(3) = New Adresse("Sonntag", "Christian", "33333", "Wuppertal")
    locAdressen(4) = New Adresse("Ademmer", "Uta", "55555", "Bad Waldholz")
    locAdressen(5) = New Adresse("Kaiser", "Wilhelm", "12121", "Ostenwesten")

    Console.WriteLine("Adressenliste:")
    Console.WriteLine(New String("=c, 40))
    DruckeAdressen(locAdressen)

    Console.WriteLine()
    Console.WriteLine("Adressenliste (sortiert nach Postleitzahl):")
    Console.WriteLine(New String("=c, 40))
    Array.Sort(locAdressen, New AdressenVergleicher(ZuVergleichen.PLZ))
    DruckeAdressen(locAdressen)
    Console.ReadLine()
End Sub
.
.
.
End Module
```

Wenn Sie das Programm nun starten, erhalten Sie folgende Ausgabe auf dem Bildschirm:

```
Adressenliste:
=====
Löffelmann, Klaus, 11111 Soest
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Kaiser, Wilhelm, 12121 Ostenwesten

Adressenliste (sortiert nach Postleitzahl):
=====
Löffelmann, Klaus, 11111 Soest
Kaiser, Wilhelm, 12121 Ostenwesten
Sonntag, Miriam, 22222 Dortmund
Sonntag, Christian, 33333 Wuppertal
Ademmer, Uta, 55555 Bad Waldholz
Heckhuis, Jürgen, 99999 Gut Uhlenbusch
```

Enumeratoren

Wenn Sie Arrays oder – wie Sie später sehen werden – Auflistungen für die Speicherung von Daten verwenden, dann müssen Sie in der Lage sein, diese Daten abzurufen. Mit *Indexern* gibt Ihnen das .NET Framework eine einfache – die einfachste – Möglichkeit: Sie verwenden eine Objektvariable und versehen sie mit einem Index, der auch durch eine andere Variable repräsentiert werden kann. Ändern Sie diese Variable, die als Index dient, können Sie dadurch programmtechnisch bestimmen, welches Element eines Arrays Sie gerade verarbeiten wollen. Typische Zählschleifen, mit denen Sie durch die Elemente eines Arrays iterieren, sind die Folge – etwa im folgenden Stil:

```
For count As Integer = 0 To Array.Length - 1
    TuWasMit(Array(count))
Next
```

HINWEIS Enumeratoren haben nichts mit Enums zu tun. Lediglich die Namen sind sich etwas ähnlich. Enums sind aufgezählte Benennungen von bestimmten Werten im Programmlisting, während Enumeratoren die Unterstützung von For/Each zur Verfügung stellen!

Wenn ein Objekt allerdings die Schnittstelle `IEnumerable` implementiert, gibt es eine elegantere Methode, die verschiedenen Elemente, die das Objekt zur Verfügung stellt, zu durchlaufen. Glücklicherweise implementiert `System.Array` die Schnittstelle `IEnumerable`, sodass Sie auf Array-Elemente mit dieser eleganten Methode – namentlich mit For/Each – zugreifen können. Ein Beispiel:

```
'Deklaration und Definition von Elementen im Char-Array
Dim locCharArray As Char() = {"V"c, "B"c, "."c, "N"c, "E"c, "T"c, " "c, _
    "r"c, "u"c, "l"c, "e"c, "s"c, "!c"}

For Each c As Char In locCharArray
    Console.Write(c)
Next
Console.WriteLine()
Console.ReadLine()
```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie im Konsolenfenster den folgenden Text:

```
VB.NET rules!
```

Enumeratoren werden von allen typdefinierten Arrays unterstützt und ebenso von den meisten Auflistungen. Enumeratoren können Sie aber auch in Ihren eigenen Klassen einsetzen, wenn Sie erlauben möchten, dass der Entwickler, der mit Ihrer Klasse arbeitet, durch Elemente mit For/Each iterieren kann.

WICHTIG Enumeratoren sind lebenswichtig für LINQ (siehe ab Kapitel 7). Nur eine Auflistungsklasse, die `IEnumerable(Of t)` implementiert, kann als Abfrageausdruck in LINQ verwendet werden. Denken Sie daran, wenn Sie eigene Auflistungsklassen entwerfen.

Benutzerdefinierte Enumeratoren durch Implementieren von IEnumerable

Enumeratoren können allerdings nicht nur für die Aufzählung von gespeicherten Elementen in Arrays oder Auflistungen eingesetzt werden. Sie können Enumeratoren auch dann einsetzen, wenn eine Klasse ihre Enumerations-Elemente durch Algorithmen zurückliefert.

Als Beispiel dafür möchte ich Ihnen zunächst einen Codeausschnitt zeigen, der nicht funktioniert – bei dem es aber in einigen Fällen wünschenswert wäre, wenn er funktionierte:

```
'Das würde nicht funktionieren:
for d as Date=#24/12/2004# to #31/12/2004#
  Console.WriteLine("Datum in Aufzählung: {0}", d)
Next d
```

Dennoch könnte es für bestimmte Anwendungen sinnvoll sein, tageweise einen bestimmten Datumsbereich zu durchlaufen. Etwa wenn Ihre Anwendung herausfinden muss, wie viele Mitarbeiter, deren Daten Ihre Anwendung speichert, in einem bestimmten Monat Geburtstag haben.

Wenn das, was wir vorhaben, nicht mit `For/Next` funktioniert, vielleicht können wir dann aber eine Klasse schaffen, die einen Enumerator zur Verfügung stellt, sodass das Vorhaben mit `For/Each` gelingt. Diese Klasse sollte beim Instanzieren Parameter übernehmen, mit denen wir bestimmen können, welcher Datumsbereich in welcher Schrittweite durchlaufen werden soll. Damit Sie mit `For/Each` durch die Elemente einer Klasse iterieren können, muss die Klasse die Schnittstelle `IEnumerable` einbinden.

Das kann sie nur, wenn sie gleichzeitig eine Funktion `GetEnumerator` zur Verfügung stellt, die erst das Objekt mit dem eigentlichen Enumerator liefert. Doch eines nach dem anderen.

BEGLEITDATEIEN

Schauen wir uns zunächst die Basisklasse an, die die Grundfunktionalität zur Verfügung stellt – Sie finden das Projekt im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\Enumerators`. Starten Sie das Programm mit `[Strg] [F5]`.

```
Public Class Datumsaufzählung
  Implements IEnumerable
  Dim locDatumsaufzähler As Datumsaufzähler

  Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
    locDatumsaufzähler = New Datumsaufzähler(StartDatum, EndDatum, Schrittweite)
  End Sub
  Public Function GetEnumerator() As System.Collections.IEnumerator _
    Implements System.Collections.IEnumerable.GetEnumerator
    Return locDatumsaufzähler
  End Function
End Class
```

Sie sehen, dass diese Klasse selbst nichts Großartiges macht – sie schafft durch die ihr übergebenen Parameter lediglich die Rahmenbedingungen und stellt die von `IEnumerable` verlangte Funktion `GetEnumerator` zur Verfügung. Die eigentliche Aufgabe wird von der Klasse `Datumsaufzähler` erledigt, die auch als Rückgabewert von `GetEnumerator` zurückgegeben wird.

Eine Instanz dieser Klasse wird bei der Instanziierung von Datumsaufzählung erstellt. Was in dieser Klasse genau passiert, zeigt der folgende Code:

```
Public Class Datumsaufzähler
    Implements IEnumerator

    Private myStartDatum As Date
    Private myEndDatum As Date
    Private mySchrittweite As TimeSpan
    Private myAktuellesDatum As Date

    Sub New(ByVal StartDatum As Date, ByVal EndDatum As Date, ByVal Schrittweite As TimeSpan)
        myStartDatum = StartDatum
        myAktuellesDatum = StartDatum
        myEndDatum = EndDatum
        mySchrittweite = Schrittweite
    End Sub

    Public Property StartDatum() As Date
        Get
            Return myStartDatum
        End Get
        Set(ByVal Value As Date)
            myStartDatum = Value
        End Set
    End Property

    Public Property EndDatum() As Date
        Get
            Return myEndDatum
        End Get
        Set(ByVal Value As Date)
            myEndDatum = Value
        End Set
    End Property

    Public Property Schrittweite() As TimeSpan
        Get
            Return mySchrittweite
        End Get
        Set(ByVal Value As TimeSpan)
            mySchrittweite = Value
        End Set
    End Property

    Public ReadOnly Property Current() As Object Implements System.Collections.IEnumerator.Current
        Get
            Return myAktuellesDatum
        End Get
    End Property

    Public Function MoveNext() As Boolean Implements System.Collections.IEnumerator.MoveNext
        myAktuellesDatum = myAktuellesDatum.Add(Schrittweite)
        If myAktuellesDatum > myEndDatum Then
            Return False
        Else
            Return True
        End If
    End Function
End Class
```

```

    End If
End Function

Public Sub Reset() Implements System.Collections.IEnumerator.Reset
    myAktuellesDatum = myStartDatum
End Sub
End Class

```

Der Konstruktor und die beiden Eigenschaftsprozeduren sollen hier nicht so sehr interessieren. Vielmehr von Interesse ist, dass diese Klasse eine weitere Schnittstelle namens `IEnumerator` einbindet, und sie stellt die eigentliche Aufzählungsfunktionalität zur Verfügung. Sie muss dazu die Eigenschaft `Current`, die Funktion `MoveNext` sowie die Methode `Reset` implementieren.

Wenn eine `For/Each`-Schleife durchlaufen wird, dann wird das aktuell bearbeitete Objekt der Klasse durch die `Current`-Eigenschaft des eigentlichen Enumerators ermittelt. Anschließend zeigt `For/Each` mit dem Aufruf der Funktion `MoveNext` dem Enumerator an, dass es auf das nächste Objekt zugreifen möchte. Erst wenn `MoveNext` mit `False` als Rückgabewert anzeigt, dass es keine weiteren Objekte mehr zur Verfügung stellen kann (oder will), ist die umgebende `For/Each`-Schleife beendet.

In unserem Beispiel müssen wir bei `MoveNext` nur dafür sorgen, dass unsere interne Datums-Zähl-Variable um den Wert erhöht wird, den wir bei ihrer Instanziierung als Schrittweite bestimmt haben. Hat die Addition der Schrittweite auf diesen Datumszähler den Endwert noch nicht überschritten, liefern wir `True` als Funktionsergebnis zurück – `For/Each` darf mit seiner Arbeit fortfahren. Ist das Datum allerdings größer als der Datums-Endwert, wird die Schleife abgebrochen – der Vorgang wird beendet.

Das Programm, das von dieser Klasse Gebrauch machen kann, lässt sich nun sehr elegant einsetzen, wie die folgenden Beispielcodezeilen zeigen:

```

Module Enumerators
    Sub Main()
        Dim locDatumsaufzählung As New Datumsaufzählung(#12/24/2004#, _
                                                    #12/31/2004#, _
                                                    New TimeSpan(1, 0, 0, 0))

        For Each d As Date In locDatumsaufzählung
            Console.WriteLine("Datum in Aufzählung: {0}", d)
        Next

    End Sub
End Module

```

Das Ergebnis sehen Sie anschließend in Form einer Datumsfolge im Konsolenfenster.

Grundsätzliches zu Auflistungen (Collections)

Arrays haben im .NET Framework einen entscheidenden Nachteil. Sie können zwar dynamisch zur Laufzeit vergrößert oder verkleinert werden, aber der Programmieraufwand dazu ist doch eigentlich recht aufwendig. Wenn Sie sich schon früher mit Visual Basic beschäftigt haben, dann ist Ihnen »Auflistung« sicherlich ein Begriff.

Auflistungen erlauben es dem Entwickler, Elemente genau wie Arrays zu verwalten. Im Unterschied zu Arrays wachsen Auflistungen jedoch mit Ihren Speicherbedürfnissen.

Damit ist aber auch klar, dass das Indizieren mit Nummern zum Abrufen der Elemente nur bedingt funktionieren kann. Wenn ein Array 20 Elemente hat, und Sie möchten das 21. Element hinzufügen, dann können Sie das dem Array nicht einfach so mitteilen. Ganz anders bei Auflistungen: Hier fügen Sie ein Element mit der Add-Methode hinzu.

Intern werden (fast alle) Auflistungen ebenfalls wie (oder besser: als) Arrays verwaltet. Rufen Sie eine neue Auflistung ins Leben, dann hat dieses Array, wenn nichts anderes gesagt wird, eine Größe von 16 Elementen. Wenn die Auflistungsklasse später, sobald Ihr Programm richtig »in Action« ist, »merkt«, dass ihr die Puste mengentechnisch ausgeht, dann legt sie ein Array nunmehr mit 32 Elementen an, kopiert die vorhandenen Elemente in das neue Array, arbeitet fortan mit dem neuen Array und tut ansonsten so, als wäre nichts gewesen.

Dieser anfängliche Load-Faktor erhöht sich bei jedem Neuanlegen des Arrays, um die Kopiervorgänge zu minimieren. Man geht einfach davon aus, dass, wenn der anfängliche Load-Faktor von 16 Elementen nicht ausreicht, 32 beim nächsten Mal auch zu wenig sind – und gerade in unserem Beispiel ist das ja auch richtig. So sind es beim zweiten Mal bereits 32 Elemente, die dazukommen, beim nächsten Mal 64 usw. bis der maximale Load-Faktor mit 2048 Elementen erreicht ist.

In etwa entspricht also die Grundfunktionsweise einer Auflistung stark vereinfacht der folgenden Klasse:

```
Class DynamicList
    Implements IEnumerable

    Protected myStepIncreaser As Integer
    Protected myCurrentArraySize As Integer
    Protected myCurrentCounter As Integer
    Protected myArray() As Object

    Sub New()
        MyClass.New(16)
    End Sub

    Sub New(ByVal StepIncreaser As Integer)
        myStepIncreaser = StepIncreaser
        myCurrentArraySize = myStepIncreaser
        ReDim myArray(myCurrentArraySize)
    End Sub

    Sub Add(ByVal Item As Object)

        '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 += myStepIncreaser

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

```

        'Elemente kopieren
        'Wichtig: Um das Kopieren müssen Sie sich,
        'anders als bei VB6, selber kümmern!
        Array.Copy(myArray, locTempArray, myArray.Length)

        'temporäres Array dem Memberarray zuweisen
        myArray = locTempArray

        'Beim nächsten Mal werden mehr Elemente reserviert!
        myStepIncreaser *= 2
    End If

    'Element im Array speichern
    myArray(myCurrentCounter) = Item

    'Zeiger auf nächstes Element erhöhen
    myCurrentCounter += 1

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 und Abfragen
Default Public Overridable Property Item(ByVal Index As Integer) As Object
    Get
        Return myArray(Index)
    End Get

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

'Liefert den Enumerator der Basis (dem Array) zurück
Public Function GetEnumerator() As System.Collections.IEnumerator Implements
System.Collections.IEnumerable.GetEnumerator
    Return myArray.GetEnumerator
End Function
End Class

```

Nun könnte man meinen, diese Vorgehensweise könnte sich zu einer Leistungsproblematik entwickeln, da alle paar Elemente der komplette Array-Inhalt kopiert werden muss.

BEGLEITDATEIEN Im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\DynamicList` finden Sie das Projekt *DynamicList*, das Ihnen das Gegenteil beweist:

Wie lange, glauben Sie, dauert es, ein Array mit 200.000 Zufallszahlen (mit Nachkommastellen) auf diese Weise anzulegen? 3 Sekunden? 2 Sekunden? Finden Sie es selbst heraus, indem Sie das Programm starten:

Anlegen von 200000 zufälligen Double-Elementen...
...in 21 Millisekunden!

Gerade mal 21 Millisekunden benötigt das Programm für diese Operation³ – beeindruckend, wie schnell Visual Basic dieser Tage ist, finden Sie nicht?

Nun muss ich Ihnen leider verraten: Die Klasse `DynamicList` werden Sie nie benötigen. Bestandteil des .NET Frameworks ist nämlich eine Klasse, die das schon kann. Sogar noch ein kleines bisschen schneller. Und: Sie hat zusätzlich noch andere Möglichkeiten, die Ihnen die selbst gestrickte Klasse nicht bietet.

Im Beispielprojekt finden Sie eine Sub `Beispiel2`, die Ihnen die gleiche Prozedur mit der Klasse `ArrayList` demonstriert:

```
Sub Beispiel2()  
    Dim locZeitmesser As New HighSpeedTimeGauge  
    Dim locAnzahlElemente As Integer = 200000  
    Dim locDynamicList As New ArrayList  
    Dim locRandom As New Random(Now.Millisecond)  
  
    Console.WriteLine("Anlegen von {0} zufälligen Double-Elementen...", locAnzahlElemente)  
    locZeitmesser.Start()  
    For count As Integer = 1 To locAnzahlElemente  
        locDynamicList.Add(locRandom.NextDouble * locRandom.Next)  
    Next  
    locZeitmesser.Stop()  
    Console.WriteLine("...in {0:##,###0} Millisekunden!", locZeitmesser.DurationInMilliseconds)  
    Console.ReadLine()  
  
End Sub
```

Dessen Geschwindigkeit ist auch nicht von schlechten Eltern:

Anlegen von 200000 zufälligen Double-Elementen...
...in 19 Millisekunden!

Im Prinzip arbeitet `ArrayList` nach dem gleichen Verfahren, das Sie in `DynamicList` kennengelernt haben. `ArrayList` verfährt auch mit dem gleichen Trick, um möglichst viel Leistung herauszuholen. Es verdoppelt die jeweils nächste Größe des neuen Arrays im Vergleich zu der vorherigen Größe des Arrays. Damit reduziert sich der Gesamtaufwand des Kopierens erheblich. Da die Methode aber Bestandteil des .NET Frameworks ist, muss sie nicht zur Laufzeit »geJITted« werden, was der Geschwindigkeit zusätzlich zugute kommt.

³ Auf einem Intel Core 2 Quad Q6600 übrigens. Lesern des Buches *Visual Studio 2005 – das Entwicklerbuch* (wie mehrfach schon erwähnt unter www.activedevelop.de kostenlos herunterladbar!) wird auffallen, dass das Ergebnis sogar eine Millisekunde langsamer ist, als auf dem Rechner, auf dem ich vor 3 Jahren das Visual Basic 2005-Buch schrieb. Sehr schön ist dabei zu sehen, dass neuere Rechner ihre Leistung kaum noch »vertikal«, über die Taktfrequenz, sondern nur noch horizontal über mehrere Prozessoren skalieren können – beim Laufenlassen dieses Beispiels wird nur einer von vier Prozessorkernen genutzt – $\frac{3}{4}$ des Prozessors liegen ungenutzt (schlimmer: in diesem Beispiel sogar unnutzbar!) brach. Um nicht nur alles sondern überhaupt mehr an Leistung aus modernen Prozessoren herauszukitzeln ist daher Multithreading-Entwicklung sehr entscheidend und sollte bei Programmportierungen – beispielsweise wenn sie ältere Borland-, MFC- oder VB6-Anwendungen auf .NET portieren wollen oder müssen – *unbedingt* berücksichtigt werden!

Im Übrigen werden die Daten der einzelnen Elemente ja nicht wirklich bewegt. Lediglich die Zeiger auf die Daten werden kopiert – vorhandene Elemente bleiben im Managed Heap an ihrem Platz. Kapitel 4 erklärt Ihnen im Abschnitt »New oder nicht New – Wieso es sich bei Objekten um Verweistypen handelt « übrigens mehr zu diesem Thema.

Wichtige Auflistungen der Base Class Library

Die BCL des .NET Frameworks enthält eine ganze Reihe von Auflistungs-Typen, von denen Sie einen der wichtigsten – `ArrayList` – schon im Einsatz gesehen haben. In diesem Abschnitt möchte ich Ihnen die wichtigsten dieser Auflistungen kurz vorstellen und darauf hinweisen, für welchen Einsatz sie am besten geeignet sind oder welche Besonderheiten Sie bei ihrem Gebrauch beachten sollten. Für eine genauere Beschreibung ihrer Eigenschaften und Methoden verwenden Sie bitte die Online-Hilfe von Visual Studio.

ArrayList – universelle Ablage für Objekte

`ArrayList` können Sie als Container für Objekte aller Art verwenden. Sie instanzieren ein `ArrayList`-Objekt und weisen ihm mithilfe seiner `Add`-Funktion das jeweils nächste Element zu. Mit der `Default`-Eigenschaft `Item` können Sie schon vorhandene Elemente abrufen oder neu definieren. `AddRange` erlaubt Ihnen, die Elemente einer vorhandenen `ArrayList` einer anderen `ArrayList` hinzuzufügen.

Mit der `Count`-Eigenschaft eines `ArrayList`-Objektes finden Sie heraus, wie viele Elemente es beherbergt.

`Clear` löscht alle Elemente einer `ArrayList`. Mit `Remove` löschen Sie ein Objekt aus der `ArrayList`, das Sie als Parameter übergeben. Wenn mehrere gleiche Objekte (die `Equals`-Methode jedes Objektes wird dabei verwendet) existieren, wird das erste gefundene Objekt gelöscht. Mit `RemoveAt` löschen Sie ein Element an einer bestimmten Position. `RemoveRange` erlaubt Ihnen schließlich, einen ganzen Bereich von `Array`-Elementen ab einer bestimmten Position im `ArrayList`-Objekt zu löschen.

`ArrayList`-Objekte können in einfache Arrays umgewandelt werden. Dabei ist jedoch einiges zu beachten: Die Elemente der `ArrayList` müssen ausnahmslos alle dem Typ des Arrays entsprechen, in den sie umgewandelt werden sollen. Die Konvertierung nehmen Sie mit der `ToArray`-Methode des entsprechenden `ArrayList`-Objektes vor. Dabei bestimmen Sie, wenn Sie in ein typdefiniertes Array (wie `Integer()` oder `String()`) umwandeln, den Grundtyp (nicht den `Arraytyp`!) als zusätzlichen Parameter. Wenn Sie ein Array in eine `ArrayList` umwandeln wollen, verwenden Sie den entsprechenden Konstruktor der `ArrayList` – die entsprechende Konstruktorroutine nimmt anschließend die Konvertierung in eine `ArrayList` vor.

HINWEIS

Bitte schauen Sie sich dazu auch das weiter unten gezeigte Listing an, insbesondere was die Konvertierungshinweise von `ArrayList`-Objekten in Arrays betrifft.

`ArrayList` implementiert die Schnittstelle `IEnumerable`. Aus diesem Grund stellt die Klasse einen Enumerator zur Verfügung, mit dem Sie mithilfe von `For/Each` durch die Elemente der `ArrayList` iterieren können. Beachten Sie dabei, den richtigen Typ für die Schleifenvariable zu verwenden. `ArrayList`-Elemente sind nicht typsicher, und eine Typ-Verletzung ist nur dann ausgeschlossen, wenn Sie genau wissen, welche Typen gespeichert sind (das nachfolgende Beispiel demonstriert diesen Fehler recht anschaulich):

Hier die Beispiele, die das gerade Gesagte näher erläutern und den Code dazu dokumentieren:

BEGLEITDATEIEN Sie finden die im Folgenden besprochenen Codeausschnitte im Verzeichnis `.\Samples\Chapter05 - ArraysCollections\CollectionsDemo`

```
Sub ArrayListDemo()  
    Dim locMännerNamen As String() = {"Jürgen", "Uwe", "Klaus", "Christian", "José"}  
    Dim locFrauenNamen As New ArrayList  
    Dim locNamen As ArrayList  
  
    'ArrayList aus vorhandenem Array erstellen.  
    locNamen = New ArrayList(locMännerNamen)  
  
    'ArrayList mit Add auffüllen.  
    locFrauenNamen.Add("Ute") : locFrauenNamen.Add("Miriam")  
    locFrauenNamen.Add("Melanie") : locFrauenNamen.Add("Anja")  
    locFrauenNamen.Add("Stephanie") : locFrauenNamen.Add("Heidrun")  
  
    'ArrayList einer anderen ArrayList hinzufügen:  
    locNamen.AddRange(locFrauenNamen)  
  
    'ArrayList in eine ArrayList einfügen.  
    Dim locHundenamen As String() = {"Hasso", "Bello", "Wauzi", "Wuffi", "Basko", "Franz"}  
    'Einfügen *vor* dem 6. Element  
    locNamen.InsertRange(5, locHundenamen)  
  
    'ArrayList in ein Array zurückwandeln.  
    Dim locAlleNamen As String()  
  
    'Vorsicht: Fehler!  
    'locAlleNamen = DirectCast(locNamen.ToArray, String())  
  
    'Vorsicht: Ebenfalls Fehler!  
    'locAlleNamen = DirectCast(locNamen.ToArray(GetType(String)), String())  
  
    'So ist es richtig.  
    locAlleNamen = DirectCast(locNamen.ToArray(GetType(String)), String())  
  
    'Repeat legt eine ArrayList aus wiederholten Items an.  
    locNamen.AddRange(ArrayList.Repeat("Dublettenname", 10))  
  
    'Ein Element im Array ändern.  
    locNamen(10) = "Fiffi"  
    'Mit der Item-Eigenschaft geht es auch:  
    locNamen.Item(13) = "Miriam"  
  
    'Löschen des ersten zutreffenden Elementes aus der Liste.  
    locNamen.Remove("Basko")  
  
    'Löschen eines Elementes an einer bestimmten Position.  
    locNamen.RemoveAt(4)  
  
    'Löschen eines bestimmten Bereichs aus der ArrayList mit RemoveRange.  
    'Count ermittelt die Anzahl der Elemente in der ArrayList.  
    locNamen.RemoveRange(locNamen.Count - 6, 5)
```

```

'Ausgeben der Elemente über die Default-Eigenschaft der ArrayList (Item).
For i As Integer = 0 To locNamen.Count - 1
    Console.WriteLine("Der Name Nr. {0} lautet {1}", i, locNamen(i).ToString)
Next

'Anderes als ein String-Objekt der ArrayList hinzufügen,
'um den folgenden Fehler "vorbereiten".
locNamen.Add(New FileInfo("C:\TEST.TXT"))

'Diese Schleife kann nicht bis zum Ende ausgeführt werden,
'da ein Objekt nicht vom Typ String mit von der Partie ist!
For Each einString As String In locNamen
    'Hier passiert irgendetwas mit dem String.
    'nicht von Interesse, deswegen kein Rückgabewert.
    einString.EndsWith("Peter")
Next
Console.ReadLine()
End Sub

```

Wenn Sie dieses Beispiel laufen lassen, dann sehen Sie zunächst die erwarteten Ausgaben auf dem Bildschirm. Doch der Programmcode erreicht nie die Anweisung `Console.ReadLine`, um auf Ihre letzte Bestätigung zu warten. Stattdessen löst er eine Ausnahme aus, etwa wie in Abbildung 5.2 zu sehen.

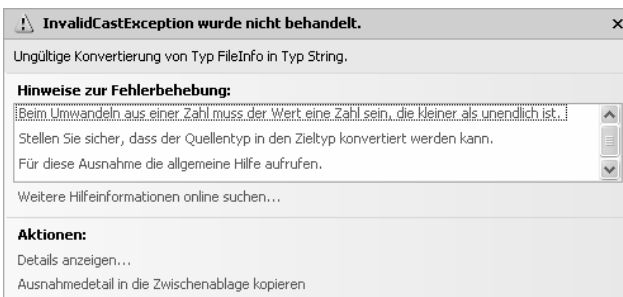


Abbildung 5.2 Achten Sie beim Iterieren mit For/Each durch eine Auflistung darauf, dass die Elemente dem Schleifenvariablentyp entsprechen, um solche Ausnahmen zu vermeiden!

Der Grund dafür: Das letzte Element in der `ArrayList` ist kein `String`. Aus diesem Grund wird die Ausnahme ausgelöst. Achten Sie deshalb stets darauf, dass die Schleifenvariable eines For/Each-Konstrukts immer den Typen entspricht, die in einer Auflistung gespeichert sind.

Sie können diesem Problem entgehen, indem Sie eine Auflistung mithilfe von *Generics* (*Generika*, auf Microsoft-Deutsch) dazu zwingen, homogen zu sein (also nur Elemente des gleichen Typs zu verarbeiten). Für den späteren Einsatz von *LINQ to Objects* (siehe Kapitel 8) ist das sogar eine Notwendigkeit.

Hashtables – für das Nachschlagen von Objekten

Hashtable-Objekte sind das ideale Werkzeug, wenn Sie eine Datensammlung aufbauen wollen, aber die einzelnen Objekte nicht durch einen numerischen Index, sondern durch einen Schlüssel abrufen wollen. Ein Beispiel soll das verdeutlichen.

Angenommen, Sie haben eine Adressverwaltung programmiert, bei der Sie einzelne Adressen durch eine Art Matchcode abrufen wollen (Kundennummer, Lieferantenummer, was auch immer). Bei der Verwendung einer `ArrayList` müssten Sie schon einigen Aufwand betreiben, um an ein Array-Element auf Basis des Matchcode-Namens zu gelangen: Sie müssten zum Finden eines Elements in der Liste für die verwendete Adressklasse eine `CompareTo`-Methode implementieren, damit die Liste mittels `Sort` sortiert werden könnte.

Anschließend könnten Sie mit `BinarySearch` das Element finden – vorausgesetzt, die `CompareTo`-Methode würde eine Instanz der Adressklasse über ihren `Matchcode`-Wert vergleichen.

`Hashtable`-Objekte vereinfachen ein solches Szenario ungemein: Wenn Sie einer `Hashtable` ein Objekt hinzufügen, dann nimmt dessen `Add`-Methode nicht nur das zu speichernde Objekt (den hinzuzufügenden Wert) entgegen, sondern auch ein weiteres. Dieses zusätzliche Objekt (das genau genommen als erster Parameter übergeben wird) stellt den Schlüssel – den `Key` – zum Wiederauffinden des Objektes dar. Sie rufen ein Objekt aus einer `Hashtable` anschließend nicht wie bei der `ArrayList` mit

```
Element = eineArrayList(5)
```

ab, sondern mit dem entsprechenden Schlüssel, etwa:

```
Element = eineHashtable("ElementKey")
```

Voraussetzung bei diesem Beispiel ist natürlich, dass der Schlüssel zuvor ein entsprechender String gewesen ist.

Queue – Warteschlangen im FIFO-Prinzip

»First in first out« (als erstes rein, als erstes raus) – nach diesem Muster arbeitet die `Queue`-Klasse der BCL. Angewendet haben Sie dieses Prinzip selbst schon sicherlich einige Male in der Praxis – und zwar immer dann, wenn Sie unter Windows mehrere Dokumente hintereinander gedruckt haben. Das Drucken unter Windows funktioniert gemäß dem Warteschlangenprinzip. Das Dokument, das als Erstes in die Warteschlange eingereicht (*enqueue* – einreihen) wurde, wird als Erstes verarbeitet und anschließend wieder aus ihr entfernt (*dequeue* – ausreihen). Aus diesem Grund verwenden Sie die Methoden `Enqueue`, um Elemente der `Queue` hinzuzufügen und `Dequeue`, um sie zurückzubekommen und gleichzeitig aus der Warteschlange zu entfernen.

BEGLEITDATEIEN

Falls Sie mit der `Queue`-Klasse experimentieren möchten, verwenden Sie dazu am besten nochmals das `CollectionsDemo`-Projekt, das Sie unter `.\Samples\Chapter05 - Arrays\Collections\CollectionsDemo` finden. Verändern Sie das Programm so, dass es die Sub `QueueDemo` aufruft, um das folgende Beispiel nachzuvollziehen:

```
Sub QueueDemo()  
  
    Dim locQueue As New Queue  
    Dim locString As String  
  
    locQueue.Enqueue("Erstes Element")  
    locQueue.Enqueue("Zweites Element")  
    locQueue.Enqueue("Drittes Element")  
    locQueue.Enqueue("Viertes Element")  
  
    'Nachschauen, was am Anfang steht, ohne es zu entfernen.  
    Console.WriteLine("Element am Queue-Anfang:" + locQueue.Peek().ToString)  
    Console.WriteLine()  
  
    'Iterieren funktioniert auch.  
    For Each locString In locQueue  
        Console.WriteLine(locString)  
    Next  
    Console.WriteLine()  
End Sub
```

```

'Alle Elemente aus Queue entfernen und Ergebnis im Konsolenfenster anzeigen.
Do
    locString = CStr(locQueue.Dequeue)
    Console.WriteLine(locString)
Loop Until locQueue.Count = 0
Console.ReadLine()
End Sub

```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element am Queue-Anfang:Erstes Element
```

```
Erstes Element
Zweites Element
Drittes Element
Viertes Element
```

```
Erstes Element
Zweites Element
Drittes Element
Viertes Element
```

Stack – Stapelverarbeitung im LIFO-Prinzip

Die Stack-Klasse arbeitet nach dem Prinzip »Last in first out« (»als letztes rein, als erstes raus«), arbeitet also genau umgekehrt zum FIFO-Prinzip der Queue-Klasse. Mit der Push-Methode schieben Sie ein Element auf den Stapel, mit Pull ziehen Sie es wieder herunter und erhalten es damit zurück. Das Element, das Sie zuletzt auf den Stapel geschoben haben, wird also mit Pull auch als Erstes wieder entfernt.

BEGLEITDATEIEN Falls Sie mit der Stack-Klasse experimentieren möchten, verwenden Sie das *CollectionsDemo*-Projekt, das Sie unter `.\Samples\Chapter05 - ArraysCollections\CollectionsDemo` finden. Verändern Sie das Programm so, dass es die Sub `StackDemo` aufruft, um das folgende Beispiel nachzuvollziehen:

```

Sub StackDemo()
    Dim locStack As New Stack
    Dim locString As String

    locStack.Push("Erstes Element")
    locStack.Push("Zweites Element")
    locStack.Push("Drittes Element")
    locStack.Push("Viertes Element")

    'Nachschauen, was oben auf dem Stapel liegt, ohne das Element zu entfernen.
    Console.WriteLine("Element zu oberst auf dem Stapel: " + locStack.Peek.ToString)
    Console.WriteLine()

    'Iterieren funktioniert auch.
    For Each locString In locStack
        Console.WriteLine(locString)
    Next
    Console.WriteLine()

```

```
'Alle Elemente vom Stack ziehen und Ergebnis im Konsolenfenster anzeigen.  
Do  
    locString = CStr(locStack.Pop)  
    Console.WriteLine(locString)  
Loop Until locStack.Count = 0  
Console.ReadLine()  
End Sub
```

Wenn Sie dieses Programm ablaufen lassen, produziert es folgende Ausgabe im Konsolenfenster:

```
Element zu oberst auf dem Stapel: Viertes Element  
  
Viertes Element  
Drittes Element  
Zweites Element  
Erstes Element  
  
Viertes Element  
Drittes Element  
Zweites Element  
Erstes Element
```

SortedList – Elemente ständig sortiert halten

Wenn Sie Elemente schon direkt nach dem Einfügen in der richtigen Reihenfolge in einer Auflistung halten wollen, dann ist die `SortedList`-Klasse das richtige Werkzeug für diesen Zweck. Allerdings sollten Sie beachten: Von allen Auflistungsklassen ist die `SortedList`-Klasse diejenige, die die meisten Ressourcen verschlingt. Für zeitkritische Applikationen sollten Sie überlegen, ob Sie Ihre Daten auch anders organisieren und stattdessen lieber auf eine unsortierte `Hashtable` oder gar auf `ArrayList` zurückgreifen können.

Der Vorteil von `SortedList` ist, dass sie quasi aus einer Mischung von `ArrayList`- und `Hashtable`-Funktionen besteht (obwohl sie algorithmisch gesehen, überhaupt nichts mit `Hashtable` zu tun hat). Sie können auf der einen Seite über einen Schlüssel, auf der anderen Seite aber auch über einen Index auf die Elemente von `SortedList` zugreifen.

Das folgende erste Beispiel zeigt den generellen Umgang mit `SortedList`.

BEGLEITDATEIEN

Sie finden dieses Projekt unter `.\Samples\Chapter05 - ArraysCollections\SortedListDemo`. Es besteht aus drei Codedateien. In der Datei `Daten.vb` finden Sie die schon bekannte `Adresse`-Klasse (bekannt, falls Sie die vorherigen Abschnitte ebenfalls durchgearbeitet haben) – allerdings in leicht veränderter Form. Der Matchcode der Zufallsadressen beginnt in dieser Version mit einer laufenden Nummer und endet mit der Buchstabenkombination des Nach- und Vornamens. Damit wird vermieden, dass eine Sortierung des Matchcodes grob auch die Adressen nach Namen und Vornamen sortiert und etwaige Nachweise eines bestimmten Programmverhaltens nicht geführt werden können.

```
Module SortedListDemo  
    Sub Main()  
        Dim locZufallsAdressen As ArrayList = Adresse.ZufallsAdressen(6)  
        Dim locAdressen As New SortedList
```

```

Console.WriteLine("Ursprungsanordnung:")
For Each locAdresse As Adresse In locZufallsAdressen
    Console.WriteLine(locAdresse)
    locAdressen.Add(locAdresse.Matchcode, locAdresse)
Next

'Zugriff per Index:
Console.WriteLine()
Console.WriteLine("Zugriff per Index:")
For i As Integer = 0 To locAdressen.Count - 1
    Console.WriteLine(locAdressen.GetByIndex(i).ToString)
Next

Console.WriteLine()
Console.WriteLine("Zugriff per Index:")
'Zugriff per Enumerator
For Each locDE As DictionaryEntry In locAdressen
    Console.WriteLine(locDE.Value.ToString)
Next
Console.ReadLine()
End Sub
End Module

```

Wenn Sie dieses Programm starten, generiert es in etwa die folgenden Ausgaben im Konsolenfenster (die Adressen werden zufällig generiert, deswegen kann die Darstellung in Ihrem Konsolenfenster natürlich wieder von der hier gezeigten abweichen).

```

Ursprungsanordnung:
00000005P1Ka: Plenge, Katrin, 26201 Liebenburg
00000004P1Ka: Plenge, Katrin, 93436 Liebenburg
00000003A1Ma: Albrecht, Margarete, 65716 Bad Waldliesborn
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim

Zugriff per Index:
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003A1Ma: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004P1Ka: Plenge, Katrin, 93436 Liebenburg
00000005P1Ka: Plenge, Katrin, 26201 Liebenburg

Zugriff per Index:
00000000AdKa: Ademmer, Katrin, 49440 Unterschleißheim
00000001LöLo: Löffelmann, Lothar, 21237 Lippetal
00000002HoBa: Hollmann, Barbara, 96807 Liebenburg
00000003A1Ma: Albrecht, Margarete, 65716 Bad Waldliesborn
00000004P1Ka: Plenge, Katrin, 93436 Liebenburg
00000005P1Ka: Plenge, Katrin, 26201 Liebenburg

```

Sie erkennen, dass die Liste in der Tat nach dem Schlüssel umsortiert wurde. Dies gilt sowohl für den Zugriff über den Index als auch über den Enumerator mit For/Each.