

19 Arrays und Auflistungen (Collections)

| | |
|-----|--|
| 540 | Grundsätzliches zu Arrays |
| 558 | Enumeratoren |
| 562 | Grundsätzliches zu Auflistungen |
| 565 | Die wichtigen Auflistungen der Base Class Library |
| 571 | Hashtables – für das Nachschlagen von Objekten |
| 587 | Queue – Warteschlangen im FIFO-Prinzip |
| 589 | Stack – Stapelverarbeitung im LIFO-Prinzip |
| 590 | SortedList – Elemente ständig sortiert halten |

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 .NET wäre nicht .NET, wenn nicht auch Arrays viel mehr Möglichkeiten bieten würden, als nur auf Daten indiziert zuzugreifen.

Arrays basieren in .NET letzten Endes wie alle anderen .NET-Klassen auf dem grundlegenden aller Datentypen, auf `Object`, und man kann sie deswegen auch als ein solches behandeln. Arrays sind also nicht nur ein simples Sprachelement in Visual Basic selbst, sondern sie gehören zur Basis des Frameworks – zur Base Class Library.

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 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.

Dieses Kapitel zeigt Ihnen, was Sie mit Arrays und den von ihnen abgeleiteten Klassen alles anstellen können.

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.

BEGLEITDATEIEN: Viele der größeren nun folgenden Beispiele sind im Projekt *Arrays* in verschiedenen *Subs* zusammengefasst. Das Projekt befindet sich im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\E - Datentypen\\Kap19\\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
```

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.

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.

Arrays als Parameter und Funktionsergebnis

Da Arrays in .NET ganz normale Objekte sind, können sie natürlich auch als Parameter in Funktionsaufrufen und als Funktionsergebnis verwendet werden. Das folgende Beispiel definiert ein Array mit String-Objekten und ruft eine Funktion auf, die das Array mit zufällig generierten Strings füllt. Eine weitere Funktion gibt das Ergebnis im Konsolenfenster aus:

```
Sub Beispiel2()  
    Dim locAnzahlStrings As Integer = 15  
    Dim locStringArray(locAnzahlStrings) As String  
    locStringArray = GeneriereStrings(locAnzahlStrings, 30)  
    DruckeStrings(locStringArray)  
    Console.ReadLine()  
End Sub  
  
Function GeneriereStrings(ByVal AnzahlStrings As Integer, ByVal LaengeStrings As Integer) As String()  
    Dim locRandom As New Random(Now.Millisecond)  
    Dim locChars(LaengeStrings - 1) As Char  
    Dim locStrings(AnzahlStrings - 1) As String  
  
    For locOutCount As Integer = 0 To AnzahlStrings - 1  
        For locInCount As Integer = 0 To LaengeStrings - 1  
            Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)  
            If locIntTemp > 26 Then  
                locIntTemp += 97 - 26  
            Else  
                locIntTemp += 65  
            End If  
            locChars(locInCount) = Convert.ToChar(locIntTemp)  
        Next  
        locStrings(locOutCount) = New String(locChars)  
    Next  
  
    Return locStrings  
End Function  
  
Sub DruckeStrings(ByVal locStringArray As String())  
    For count As Integer = 0 To locStringArray.Length - 1  
        If Not (locStringArray(count) Is Nothing) Then  
            Console.WriteLine(locStringArray(count))  
        Else  
            Console.WriteLine(" ")  
        End If  
    Next  
End Sub
```

```

        Console.WriteLine("--- NOTHING ---")
    End If
Next
End Sub

```

Wenn Sie dieses Beispiel laufen lassen, sehen Sie im Konsolenfenster eine Reihe von Zeichenketten, die in etwa den folgenden entsprechen:

```

SfwMyCKJJckzKp0sPJXHPxZfRwXqxB
[DKZJpJGIuLRiAKLhmfDThqBUGRvC
fTMIbhp1g[jKBdwyQZVGQqFSYHWUXp
jrLGPsf[zdvVbMwUSEuEFhxpmpPuju
LVOrrrzw0Esq[r1UosWFqS[TKCvWUb
dLrETAiLVFVWqqLPHFESAXYffTvvHp
xdcnbizWZ1neXRkUckIVvqSC[GnM{[
FnEgIsuDPeQ1gTFX{Hv1RLnmvHL[NV
vnWzg{[rJsZYVeFJJzXEHgROJuAT1B
hUqujcPingxJyCmtpJg1yMDJOPWIpM
{HFIMFicvdubMrHyhPCRfSnADURhgU
GHsBgqEqrHKONDrXBMCQiHFZHIUFFr
oBwcXnjKLRurYwGrejJgUEfPmzCUUY
Wym{TJSmgEorBmWrjKSrelwYXkXhqY
1PqnwvFYdxZsDFtdDttvQtBeukC0dj

```

Sie sehen an diesem Beispiel, dass Sie Arrays als Parameter wie ganz normale andere Objekte in .NET verwenden können – doch dazu mehr in einem der nächsten Abschnitte.

Sie sehen an diesem Beispiel auch, dass sich Arrays dynamisch dimensionieren lassen – die Größe eines Arrays muss Ihnen also nicht schon zur Entwurfszeit Ihres Programms bekannt sein. Sie können – und das macht den Einsatz von Arrays unter .NET so flexibel – Arrays mit variablen Werten (im wahrsten Sinne des Wortes) zur Laufzeit anlegen.

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 demonstrieren und verdeutlichen:

```

Sub Beispiel3()

    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("=", 40))
    DruckeStrings(locStringArray)

```

```

'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 typsicheren Neudimensionieren eines Arrays. Wie Sie beim ersten Array-Beispiel gesehen haben, spielt es natürlich keine Rolle, ob Sie eine Array-Variable 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 Unter-routine 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 Umdimensionieren 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 auch 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` für eine Objektvariable aufgerufen, wird eine komplett neue Klasseninstanz eines Arrays erstellt. Ein entsprechender Speicherbereich wird 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 (soweit wie möglich, falls das neue Array über weniger Elemente verfügt) 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.

Arrays sind .NET-Objekte – das hat den einen oder anderen Vorteil!

Jedes typsichere² Array in .NET ist von `System.Array` abgeleitet. Eine Objektvariable vom Typ `System.Array` kann also ebenfalls als Zeiger auf ein Array dienen. Auf den ersten Blick bringt das nicht viel, denn ein `System.Array` hat keinen *Indexer* (keine parametrisierte *Default*-Eigenschaft), sodass es zunächst so scheint, dass Sie an die Elemente nicht mehr herankommen. Aber: `System.Array` verfügt über einen *Enumerator* (eine Funktionalität, die es gestattet, mit *For/Each* auf die Elemente zuzugreifen). Und: `GetValue` und `SetValue` erlauben das Zugreifen auf die Array-Elemente. Gerade wenn Sie den Einsatz von generischen Klassen vermeiden möchten, können Sie mit ein paar Handgriffen und

² Typdefiniert in diesem Zusammenhang bedeutet: Jedes Element eines Arrays ist vom selben Typ, und dieser Typ wird bei der Definition des Arrays festgelegt. Das Gegenteil dazu wären Arrays, bei denen jedes Element eines Arrays ein anderes sein kann und in einem Objekt geboxed ist.

Tricks sich diesem Konzept der generischen Programmierung wenigstens bis auf ein paar Meter nähern, und eine Klasse typischer vererbbar aber eben nicht generisch formulieren.

Mal angenommen, Sie verarbeiten alle möglichen numerischen Daten in einem Anwendungsprogramm – lassen wir dies beispielsweise eine Statistik-Anwendung sein – und Sie benötigen für die verschiedensten Datentypen eine Routine, die das Maximum aus einer Reihe von Zahlen ermittelt. Sie möchten das Rad aber nicht ständig neu erfinden und für jeden Datentyp, den Sie benötigen, eine Max-Funktion entwerfen. Idealerweise sollten Sie über eine Funktion verfügen, die folgenden Beispielcode erlaubt:

```
Sub Beispiel4()  
    Dim locDoubles(100) As Double  
    Dim locIntegers(100) As Integer  
    Dim locDecimals(100) As Decimal  
    Dim locRandom As New Random(Now.Millisecond)  
  
    'Jedes Array mit 101 Zufallszahlen füllen.  
    '(Nicht vergessen: 100 ist das höchste Element, nicht die Länge ;-)  
    For count As Integer = 0 To 100  
        locDoubles(count) = locRandom.NextDouble * locRandom.Next  
        locDecimals(count) = Convert.ToDecimal(locRandom.NextDouble * locRandom.Next)  
        locIntegers(count) = locRandom.Next  
    Next  
  
    Console.WriteLine("Das größte Element des Double-Arrays war {0}", Cdbl(Max(locDoubles)))  
    Console.WriteLine("Das größte Element des Integer-Arrays war {0}", Cint(Max(locIntegers)))  
    Console.WriteLine("Das größte Element des Decimal-Arrays war {0}", Cdec(Max(locDecimals)))  
    Console.ReadLine()  
End Sub
```

Sie sehen: Die Max-Funktion gibt es nur einmal, und sie nimmt beliebige Arrays entgegen, ganz gleich, welchen spezielleren, typdefinierten Array-Datentyp sie besitzen.

Möglich wird das, weil wir uns in der aufgerufenen Funktion nicht auf ein Array eines näher spezifizierten Typs fixiert haben, sondern uns ganz allgemein der Basisklassen aller typdefinierten Arrays bedienen, nämlich `System.Array`³:

```
Enum Vergleich  
    Kleiner = -1  
    Gleich = 0  
    Größer = 1  
End Enum  
  
Function Max(ByVal Array As System.Array) As IComparable  
  
    'Leeres Array-Objekt, dann beenden.  
    If Array.Length = 0 Then  
        Return Nothing  
    End If
```

³ Ich füge hier bewusst den *Namespace*-Namen *System* in die Bezeichnung ein, um den Typ *System.Array* vom Begriff *Array* zu unterscheiden.

```

'Kann nur vergleichbare Elemente vergleichen,
'aber alle primitiven Datentypen sind glücklicherweise
'vergleichbar, und sie implementieren IComparable.
Dim locICElement As IComparable

'Erstes Element als Basis holen.
locICElement = DirectCast(Array.GetValue(0), IComparable)
If locICElement Is Nothing Then
    'Implementiert nicht IComparable, dann Abbrechen.
    Return Nothing
End If

For Each locICSchleifenElement As IComparable In Array
    If locICSchleifenElement Is Nothing Then
        Return Nothing
    End If

    If locICSchleifenElement.CompareTo(locICElement) = Vergleich.Größer Then
        locICElement = locICSchleifenElement
    End If
Next
Return locICElement

End Function

```

Die Enum am Anfang der Prozedur hilft lediglich, den Code leichter zu lesen, und sie dient als Parameter für die später folgende `CompareTo`-Methode.

Interessant wird es erst in der Funktion `Max`, die das eigentliche Maximum des Arrays ermittelt. Sie nimmt – wie schon gesagt – ein `System.Array` entgegen, und damit wird der Weg für alle beliebigen Arrays frei. Zurück – und das ist das Interessante bei dieser Funktion – liefert sie einen Wert vom Typ `IComparable`. Und da alle primitiven Datentypen diese Schnittstelle implementieren, kann die Funktion indirekt auch jeden Datentyp zurückliefern. Die aufrufende Instanz muss den `IComparable`-Rückgabewert lediglich mit einer Umwandlungsanweisung (`DirectCast` für Objekte oder `CType` bzw. `Cxxx` für Wertetypen) wieder in den ihr bekannten Typ zurückwandeln.

Um den größten Wert in einem Array aus Elementen zu finden, muss die Prozedur die verschiedenen Werte miteinander vergleichen können. Idealerweise ist die Vergleichsfunktion genau die Funktion, die `IComparable` den sie einbindenden Klassen vorschreibt. Die `Max`-Prozedur kann sich also blind darauf verlassen, dass ein gültiges `IComparable`-Objekt auch die Vergleichsmethode `CompareTo` anbietet. Genau diese Tatsache macht sie sich zunutze, wenn sie mit `For/Each` durch die Schleife iteriert. Es interessiert sie gar nicht, *welche* Objekttypen sie vergleicht. Sind es `Decimals`, dann wird eben `CompareTo` eines `Decimal`-Datentyps aufgerufen, bei `Integer` die `CompareTo`-Methode des `Integer`-Datentyps. Schnittstellen machen es einmal mehr möglich!

Kleiner Hinweis: Sollte das Array-Element, das gerade verarbeitet wird, die `IComparable`-Schnittstelle nicht einbinden, dann wird die Objektvariable, die das Element hält, zu `Nothing`. In diesem Fall bricht die `Max`-Funktion die Verarbeitung ab und liefert auch `Nothing` als Funktionsergebnis zurück. Die aufrufende Instanz des Beispielprogramms prüft aber nicht auf `Nothing` als Rückgabergebnis, weil sie ja weiß, dass alle übergebenden Arrays `IComparable`-Elemente aufweisen und der `Nothing`-Fall niemals eintreten kann.

HINWEIS: Das soll hier im Beispiel reichen. Im wirklichen Programmiererleben sollte man gerade zu große Arrays, zu kleine Arrays oder komplett fehlende Rückgaben (*Nothing*) wie hier prüfen und diese adäquat behandeln.

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 quasi zu Fuß mit Daten vorbelegen, die das Programm fest vorgibt.

Im Gegensatz zu Visual Basic 6.0, bei der diese Prozedur eine endlose Quälerei des Codehackens war, indem Sie jedem einzelnen Element mit dem Zuweisungsoperator den entsprechenden Wert zuweisen mussten, geht es in Visual Basic .NET viel eleganter und schneller, wie das folgende Beispiel zeigt:

```
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 Definitionsvorgehensweise: 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 des Arrays die Anzahl der zu verwaltenden Elemente jeder Dimension durch Komma getrennt angeben:

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

Möchten Sie die Anzahl der zu verwaltenden Elemente pro Dimension 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 halten, und diese Zuordnung lässt sich auch im Nachhinein noch ändern.

Verschachtelte Arrays definieren Sie, indem Sie die Klammernpaare mit der entsprechenden Array-Dimension hintereinander schreiben – anders als bei mehrdimensionalen Arrays, bei denen die Dimensionen in einem Klammernpaar 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("Außeres 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, und 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(locNamen)
    DruckeStrings(locNamen)
    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 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(locNamen)  
DruckeStrings(locNamen)  
Console.ReadLine()
```

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

```
Absteigend sortiert:  
Uwe  
Ute  
Thomas  
.  
.  
.  
Anja  
Andrea
```

Durchsuchen eines sortierten Arrays mit `Array.BinarySearch`

Auch bei der Suche nach bestimmten Elementen eines Arrays ist Ihnen das 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 kann die Funktion 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 Indextbereiche beschränkt. IntelliSense und die Online-Hilfe geben hier genauere Hinweise für die Verwendung.

Implementierung von Sort und BinarySearch für eigene Klassen

Das 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 *VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\Comparer01*.

```
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

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

```

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

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
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 Framework eine Ausnahme aus, etwa wie in Abbildung 19.1 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 *Comparers* verglichen und im Endeffekt sortiert werden.

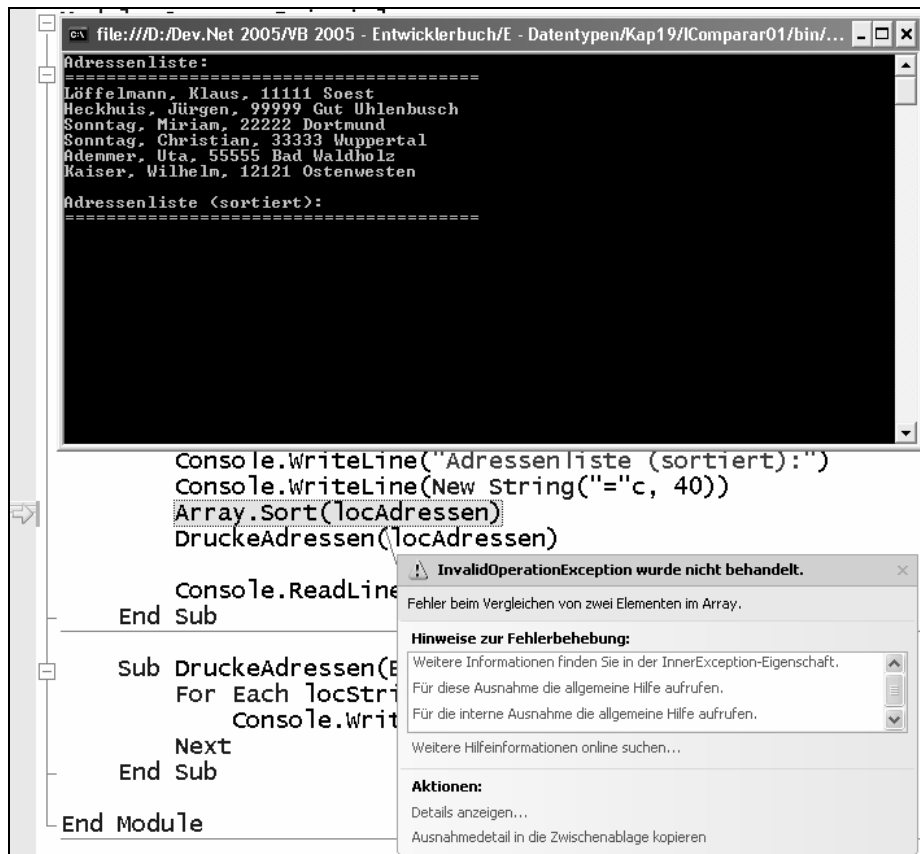


Abbildung 19.1: Wenn Sie das Programm starten, löst es eine Ausnahme aus, sobald die *Sort*-Methode der *Array*-Klasse erreicht ist

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 der Klasse miteinander verglichen werden können, ist vergleichsweise simpel. Das Einbinden erfordert zusätzlich lediglich das Vorhandensein einer *CompareTo*-Methode, die die aktuelle Instanz der Klasse mit einer weiteren vergleicht. Da durch den Einsatz von *IComparable* keine Möglichkeit besteht festzulegen, welches Datenfeld das Vergleichskriterium sein soll, wird unser Kriterium eine Zeichenkette sein, die aus Name, Vorname, 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. Die Fehlermeldung, die wir in der Ursprungsversion unseres Beispiels in Abbildung 19.1 gesehen habe, leistete das beispielsweise nicht. Die Fehlermeldung verwirrt mehr, als sie den Zusammenhang aufklärt. 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 bestimmt, 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:

BEGLEITDATEIEN: Sie finden das erweiterte Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\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 'Adressenvergleich' 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("=", 40))
        DruckeAdressen(1ocAdressen)

        Console.WriteLine()
        Console.WriteLine("Adressenliste (sortiert nach Postleitzahl):")
        Console.WriteLine(New String("=", 40))
        Array.Sort(1ocAdressen, New AdressenVergleicher(ZuVergleichen.PLZ))
        DruckeAdressen(1ocAdressen)
        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 .NET eine einfache 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.WriteLine(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.

Benutzerdefinierte Enumeratoren mit `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 `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\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 Instanzierung 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

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 Instanzierung 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

Arrays haben in .NET einen entscheidenden Nachteil. Sie können zwar dynamisch zur Laufzeit vergrößert oder verkleinert werden, aber der Programmieraufwand dazu ist doch eigentlich recht aufwändig. 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 2. 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, die Ihnen in den vergangenen Kapiteln schon mehrfach begegnet ist:

```

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 *VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\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 20 Millisekunden!

```

Gerade mal 20 Millisekunden benötigt das Programm für diese Operation⁴ – 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 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:

⁴ Auf einem Athlon 4400+.

```

Sub Beispiel2()
    Dim locZeitmesser As New HighSpeedTimeGauge
    Dim locAnzahlElemente As Integer = 50000
    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 50000 zufälligen Double-Elementen...
...in 8 Millisekunden!

```

Im Prinzip arbeitet ArrayList nach dem gleichen Verfahren, das Sie in DynamicList kennen gelernt 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 Frameworks ist, muss sie nicht zur Laufzeit »geJITted« werden, was der Geschwindigkeit zusätzlich zugute kommt.

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.

Die wichtigen Auflistungen der Base Class Library

Die BCL des 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 wichtigen 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 der Remove-Methode 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 die Konvertierung in eine ArrayList anschließend vor.

HINWEIS: Bitte schauen Sie sich dazu auch das weiter unten gezeigte Listing an, insbesondere was die Konvertierungshinweise von ArrayList-Objekte 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:

HINWEIS: Sie finden die im Folgenden besprochenen Codeausschnitte im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\CollectionsDemo\`.

```
Sub ArrayListDemo()  
    Dim locMännerNamen As String() = {"Jürgen", "Uwe", "Klaus", "Christian", "José"}  
    Dim locFrauenNamen As New ArrayList  
    Dim locNamen As New ArrayList  
  
    'ArrayList aus vorhandenem Array/Arraylist erstellen.  
    locNamen = New ArrayList(locMännerNamen)  
  
    'ArrayList mit Add definieren.  
    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 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 mit der Item-Eigenschaft 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 19.2 zu sehen.

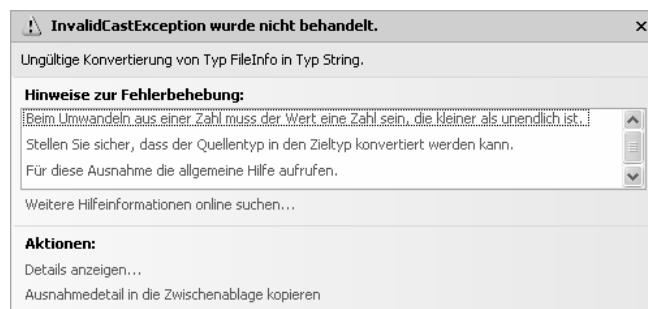


Abbildung 19.2: Achten Sie beim Iterieren mit *For/Each* durch eine Auflistung darauf, dass die Elemente dem Schleifenvariablentyp entsprechen, um solche Ausnahmen zu vermeiden!

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.

Typsichere Auflistungen auf Basis von `CollectionBase`

Um den im vorherigen Abschnitt aufgetretenen Fehler definitiv zu vermeiden, sollte eine Auflistung in der Lage sein, nur Elemente bestimmten Typs zu definieren und zurückzuliefern. Die `ArrayList`-Klasse kann diese Arbeit nicht leisten, weil sie – beispielsweise mit der `Add`-Methode – Objects entgegennimmt und damit für alle Objekttypen zugänglich ist.

Gerade wenn Sie Softwareentwicklung im Team betreiben, ist es sicherer, dass Sie den Entwicklern, die mit Ihren Klassen arbeiten, nur solche zur Verfügung stellen, die narrensicher zu bedienen sind. Haben Sie beispielsweise einen Klassensatz entwickelt, der Adressendetails in einer `Adresse`-Klasse und mehrere Adressen in einer `Adressen`-Auflistung verwaltet, dann sollte Letztere auch nur Daten vom Typ `Adresse` entgegennehmen. Sie sollte damit typsicher sein.

Mit dem Framework der Version 2.0 gibt es grundsätzlich zwei Möglichkeiten, dabei Typsicherheit zu garantieren. Sie können die Klasse von der generischen Klasse `Collection(Of T)` bzw. `List(Of T)` ableiten – und haben die Aufgabe erledigt. Das anschließende ► Kapitel 20 verrät Ihnen mehr zu dieser Möglichkeit.

Um Kompatibilität mit Framework-Versionen zu wahren, die generische Klassen noch nicht unterstützen (1.0, 1.1) und um ein weiteres, wirklich wichtiges Beispiel für die klassische OOP-Programmierung kennen zu lernen, lesen Sie den folgenden Abschnitt.

Das Framework bietet zu diesem Zweck eine abstrakte Klasse namens `CollectionBase` als Vorlage an, die Sie für diese Zwecke ableiten und erweitern können.

BEGLEITDATEIEN: Sie finden die im Folgenden besprochenen Codeauszüge im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\TypeSafeCollections\` im Verzeichnis der CD zum Buch.

Der folgende Code zeigt, wie die `Item`-Methode und die `Add`-Methode in einer benutzerdefinierten Auflistung realisiert sind:

```

Public Class Adressen
    Inherits CollectionBase

    Public Overridable Function Add(ByVal Adr As Adresse) As Integer
        Return MyBase.List.Add(Adr)
    End Function

    Default Public Overridable Property Item(ByVal Index As Integer) As Adresse
        Get
            Return DirectCast(MyBase.List(Index), Adresse)
        End Get
        Set(ByVal Value As Adresse)
            MyBase.List(Index) = Value
        End Set
    End Property
End Class

```

Das entsprechende Beispielprogramm, das die Klasse testet, sieht folgendermaßen aus:

```

Module TypesafeCollections

    Sub Main()
        Dim locAdressen As New Adressen
        Dim locAdresse As New Adresse("Christian", "Sonntag", "99999", "Trinken")
        Dim locAndererTyp As New FileInfo("C:\Test.txt")

        'Kein Problem:
        locAdressen.Add(locAdresse)

        'Schon der Editor mault!
        'locAdressen.Add(locAndererTyp)

        'Auch kein Problem.
        locAdresse = locAdressen(0)

        For Each eineAdresse As Adresse In locAdressen
            Console.WriteLine(eineAdresse)
        Next
        Console.ReadLine()
    End Sub

End Module

```

Sie sehen anhand des Testprogramms, dass Typsicherheit in dieser Auflistung gewährleistet ist. Allerdings sollte dieses Programm, wenn Sie das Klassen-Kapitel dieses Buches aufmerksam studiert haben, auch Fragen aufwerfen: Wenn Sie sich die Beschreibung zu `CollectionBase` in der Online-Hilfe ansehen, finden Sie dort folgenden Prototyp:

```

<Serializable>
MustInherit Public Class CollectionBase
    Implements IList, ICollection, IEnumerable
    .
    .
    .

```

CollectionBase bindet unter anderem die Schnittstelle IList ein. Die Schnittstelle IList schreibt vor, dass eine Add-Funktion mit der folgenden Signatur in der Klasse implementiert sein muss, die sie implementiert:

```
Function Add(ByVal value As Object) As Integer
```

Erkennen Sie die Unregelmäßigkeit? Unsere Klasse erbt von CollectionBase, CollectionBase bindet IList ein, IList verlangt die Methode Add, in unserer Klasse müsste demzufolge eine Add-Methode vorhanden sein, die wir zu überschreiben hätten! Aber sie ist es nicht, und das ist auch gut so, denn: Wäre sie vorhanden, dann müssten wir ihre Signatur zum Überschreiben übernehmen. Als Parameter nimmt sie aber ein Object entgegen – unsere Typsicherheit wäre dahin. Alternativ könnten wir sie überladen, aber in diesem Fall könnte man ihr dennoch ein Object übergeben – wieder wäre es aus mit der Typsicherheit.

Wir kommen dem Geheimnis auf die Spur, wenn wir uns die Add-Funktion unserer neuen Adressen-Auflistung ein wenig genauer anschauen. Dort heißt es:

```
Public Overridable Function Add(ByVal Adr As Adresse) As Integer
    Return MyBase.List.Add(Adr)
End Function
```

MyBase greift auf die Basisklasse zurück und verwendet letzten Endes die Add-Funktion des Objektes, das die List-Eigenschaft zurückliefert, um das hinzuzufügende Element weiterzureichen. Was aber macht List? Welches Objekt liefert es zurück? Um Sie vielleicht zunächst komplett zu verwirren: List liefert die Instanz unserer Klasse zurück⁵ – und wir betreiben hier Polymorphie in Vollendung!

Schauen wir, was die Online-Hilfe von Visual Studio zur Beschreibung von List zu sagen hat: »Ruft eine IList mit der Liste der Elemente in der CollectionBase-Instanz ab.« – Wow, das ist informativ!

Der ganze Umstand wird klar, wenn Sie sich das folgende Konstrukt anschauen. Behalten Sie dabei im Hinterkopf, dass eine Möglichkeit gefunden werden muss, die Add-Funktion einer Schnittstelle in einer Klasse zu implementieren, ohne einer sie einbindenden Klasse die Möglichkeit zu nehmen, eine Add-Funktion mit einer ganz anderen Signatur zur Verfügung zu stellen:

```
Interface ITest
    Function Add(ByVal obj As Object) As Integer
End Interface

MustInherit Class ITestKlasse
    Implements ITest

    Private Function ITestAdd(ByVal obj As Object) As Integer Implements ITest.Add
        Trace.WriteLine("ITestAdd:" + obj.ToString)
    End Function

End Class
```

⁵ Dieses Konstrukt erinnerte mich spontan an eine Star-Trek-Voyager-Episode, in der Tom Paris mit B'Elanna Torres das Holodeck besucht, um mit ihr in einem holografisch projizierten Kino einen »altertümlichen« Film mit 3-D-Brille zu sehen. Ihr Kommentar dazu: „Lass mich das mal klarstellen: Du hast diesen Aufwand betrieben, um eine dreidimensionale Umgebung so zu programmieren, das sie ein zweidimensionales Bild projiziert, und nun bittest Du mich, diese Brille zu tragen, damit es wieder dreidimensional aussieht?« ...

```

Class ITestKlasseAbleitung
  Inherits ITestKlasse

  Public ReadOnly Property Test() As ITest
    Get
      Return DirectCast(Me, ITest)
    End Get
  End Property

  Public Sub Add(ByVal TypeSafe As Adresse)
    Test.Add(TypeSafe)
  End Sub
End Class

```

Die Schnittstelle dieser Klasse schreibt vor, dass eine Klasse, die diese Schnittstelle einbindet, eine Add-Funktion implementieren muss. Die abstrakte Klasse ITestKlasse bindet diese Schnittstelle auch ordnungsgemäß ein – allerdings stellt sie die Funktion nicht der Öffentlichkeit zur Verfügung; die Funktion ist dort nämlich als privat definiert. Außerdem – und das ist der springende Punkt – nennt sich die Funktion nicht Add, sondern ITestAdd – möglich wird das, da Schnittstelleneinbindungen in Visual Basic grundsätzlich explizit funktionieren, genauer gesagt durch das Schlüsselwort Implements am Ende der Funktionsdeklaration.

Halten wir fest: Wir haben nun eine saubere Schnittstellenimplementierung, *und* wir können die der Schnittstelle zugeordnete Funktion von außen nicht sehen. Wie rufen wir die Funktion IListAdd der Basisklasse ITestKlasse dann überhaupt auf? Der Rätsels Lösung ist: Die Funktion ist nicht ganz so privat, wie sie scheint. Denn wir können über eine Schnittstellenvariable auf sie zugreifen. Wenn wir die eigene Instanz der Klasse in eine Schnittstellenvariable vom Typ ITest casten, dann können wir über die Schnittstellenfunktion ITest.Add dennoch auf die private Funktion ITestAdd der Klasse ITestKlasse zugreifen – und wir sind am Ziel!

Zu unserer Bequemlichkeit stellt die Klasse ITestKlasse bereits eine Eigenschaft (Property Test) zur Verfügung, die die aktuelle Klasseninstanz als ITest-Schnittstelle zurückliefert. Wir können uns dieser also direkt bedienen.

Genau das Gleiche haben wir in unserer aus CollectionBase entstandenen Klasse Adressen gemacht – um zum alten Beispiel zurückzukommen. Die List-Eigenschaft der Klasse CollectionBase entspricht der Test-Eigenschaft der Klasse ITestKlasse unseres Erklärungsbeispiels.

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.

Anwenden von Hashtables

BEGLEITDATEIEN: Im Verzeichnis `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\HashtableDemo01\` des Verzeichnisses der CD zum Buch finden Sie ein Beispielprojekt, das den Einsatz der Hashtable-Klasse demonstrieren soll. Es enthält eine Klasse, die eine Datenstruktur – eine Adresse – abbildet. Eine statische Funktion erlaubt darüber hinaus, eine beliebige Anzahl von Zufallsadressen zu erstellen, die zunächst in einer ArrayList gespeichert werden. Eine solche mit zufälligen Daten gefüllte ArrayList soll zum Experimentieren mit der Hashtable-Klasse dienen. Der Vollständigkeit halber (und des späteren einfacheren Verständnisses) möchte ich Ihnen diese Adresse-Klasse kurz vorstellen – wenn auch in verkürzter Form):

WICHTIG: Sie benötigen mindestens 1 GByte-Arbeitsspeicher, damit Sie die folgenden Hashtable-Beispiele nachvollziehen können. Die Menge des zu reservierenden Speichers für das Beispiel ist deshalb so groß, weil durch die hohen Prozessorgeschwindigkeiten eine genügend große Anzahl von Elementen vorhanden sein muss, um überhaupt Geschwindigkeitsvergleiche anstellen zu können.

```
Public Class Adresse
    'Member-Variablen, die die Daten halten:
    Protected myMatchcode As String
    Protected myName As String
    Protected myVorname As String
    Protected myPLZ As String
    Protected myOrt As String

    'Konstruktor - legt eine neue Instanz an.
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
        ByVal Plz As String, ByVal Ort As String)
        myMatchcode = Matchcode
        myName = Name
        myVorname = Vorname
        myPLZ = Plz
        myOrt = Ort
    End Sub
```

```

'Mit Region ausgeblendet:
'die Eigenschaften der Klasse, um die Daten offen zu legen.
#Region "Eigenschaften"
.
.
.
#End Region
Public Overrides Function ToString() As String
    Return Matchcode + ": " + Name + ", " + Vorname + ", " + PLZ + " " + Ort
End Function

Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As ArrayList

    Dim locArrayList As New ArrayList(Anzahl)
    Dim locRandom As New Random(Now.Millisecond)

    Dim locNachnamen As String() = {"Heckhuis", "Löffelmann", "Thiemann", "Müller", _
        "Meier", "Tiemann", "Sonntag", "Ademmer", "Westermann", "Vüllers", _
        "Hollmann", "Vielstedde", "Weigel", "Weichel", "Weichelt", "Hoffmann", _
        "Rode", "Trouw", "Schindler", "Neumann", "Jungemann", "Hörstmann", _
        "Tinoco", "Albrecht", "Langenbach", "Braun", "Plenge", "Englisch", _
        "Clarke"}

    Dim locVornamen As String() = {"Jürgen", "Gabriele", "Uwe", "Katrin", "Hans", _
        "Rainer", "Christian", "Uta", "Michaela", "Franz", "Anne", "Anja", _
        "Theo", "Momo", "Katrin", "Guido", "Barbara", "Bernhard", "Margarete", _
        "Alfred", "Melanie", "Britta", "José", "Thomas", "Daja", "Klaus", "Axel", _
        "Lothar", "Gareth"}

    Dim locStädte As String() = {"Wuppertal", "Dortmund", "Lippstadt", "Soest", _
        "Liebenburg", "Hildesheim", "München", "Berlin", "Rheda", "Bielefeld", _
        "Braunschweig", "Unterschleißheim", "Wiesbaden", "Straubing", _
        "Bad Waldliesborn", "Lippetal", "Stirpe", "Erwitte"}

    For i As Integer = 1 To Anzahl
        Dim locName, locVorname, locMatchcode As String
        locName = locNachnamen(locRandom.Next(locNachnamen.Length - 1))
        locVorname = locVornamen(locRandom.Next(locNachnamen.Length - 1))
        locMatchcode = locName.Substring(0, 2)
        locMatchcode += locVorname.Substring(0, 2)
        locMatchcode += i.ToString("00000000")
        locArrayList.Add(New Adresse( _
            locMatchcode, _
            locName, _
            locVorname, _
            locRandom.Next(99999).ToString("00000"), _
            locStädte(locRandom.Next(locStädte.Length - 1))))
    Next
    Return locArrayList
End Function

```

```

Shared Sub AdressenAusgeben(ByVal Adressen As ArrayList)
  For Each Item As Object In Adressen
    Console.WriteLine(Item)
  Next
End Sub

```

End Class

Die eigentliche Klasse zum Speichern einer Adresse ist Kinderkram. Wichtig ist, dass Sie wissen, welche besondere Bewandnis es mit dem Matchcode einer Adresse hat. Ein Matchcode ist in einem Satz von Adressen immer eindeutig. Die Prozedur in diesem Beispiel, die zufällige Adressen erzeugt, stellt das sicher. Der Matchcode setzt sich aus den ersten beiden Buchstaben des Nachnamens, den ersten beiden Buchstaben des Vornamens und einer fortlaufenden Nummer zusammen. Würden Sie 15 verschiedene Zufallsadressen mit diesem Code

```

Sub AdressenTesten()

  '15 zufällige Adressen erzeugen.
  Dim locDemoAdressen As ArrayList = Adresse.ZufallsAdressen(15)

  'Adressen im Konsolenfenster ausgeben.
  Console.WriteLine("Liste mit zufällig erzeugten Personendaten")
  Console.WriteLine(New String("=c, 30))
  Adresse.AdressenAusgeben(locDemoAdressen)

End Sub

```

erstellen und ausgeben, sähen Sie etwa folgendes Ergebnis im Konsolenfenster:

```

Liste mit zufällig erzeugten Personendaten
=====
HeMo00000001: Heckhuis, Momo, 06549 Straubing
SoGu00000002: Sonntag, Guido, 21498 Liebenburg
ThAl00000003: Thiemann, Alfred, 51920 Bielefeld
HöJü00000004: Hörstmann, Jürgen, 05984 Liebenburg
TiMa00000005: Tiemann, Margarete, 14399 München
TiAn00000006: Tiemann, Anja, 01287 Dortmund
ViGu00000007: Vielstedde, Guido, 72762 Wuppertal
RoMe00000008: Rode, Melanie, 94506 Hildesheim
TiDa00000009: Tiemann, Daja, 54134 Lipstadt
BrJo00000010: Braun, José, 14590 Soest
WeJü00000011: Westermann, Jürgen, 83128 Wuppertal
HeKa00000012: Heckhuis, Katrin, 13267 Bad Waldliesborn
TrJü00000013: Trouw, Jürgen, 54030 Lipstadt
PlGa00000014: Plenge, Gabriele, 97702 Braunschweig
WeJü00000015: Weichel, Jürgen, 39992 Unterschleißheim

```

Falls Sie sich nun fragen, wieso ich Ihnen soviel Vorgeplänkel erzähle: Es ist wichtig für die richtige Anwendung von Hashtable-Objekten und das richtige Verständnis, wie Elemente in Hashtables gespeichert werden. Denn wenn Sie ein Objekt in einer Hashtable speichern, muss der Schlüssel eindeutig sein – anderenfalls hätten Sie nicht die Möglichkeit, wieder an alle Elemente heranzukommen. (Welches von zwei Elementen sollte die Hashtable schließlich durch einen Schlüssel indiziert zurückliefern, wenn die Schlüssel die gleichen wären?)

Verarbeitungsgeschwindigkeiten von Hashtables

Jetzt lassen Sie uns eine Hashtable in Aktion sehen und schauen, was sie zu leisten vermag. Aus den Elementen der ArrayList, die uns die ZufallsAdressen-Funktion liefert, bauen wir eine Hashtable mit nicht weniger als 500.000 Elementen auf. Gleichzeitig erzeugen wir ein weiteres Array mit 50 zufälligen Elementen der ArrayList und merken uns deren Matchcode. Diese Matchcodes verwenden wir anschließend, um uns Elemente aus der Liste herauszupicken und messen dabei die Zeit, die das Zugreifen benötigt. Das Programm dazu sieht folgendermaßen aus:

```
Module HashtableDemo
```

```
Sub Main()
```

```
    'AdressenTesten()  
    'Console.ReadLine()  
    'Return
```

```
    Dim locAnzahlAdressen As Integer = 1000000  
    Dim locZugriffsElemente As Integer = 2  
    Dim locMessungen As Integer = 3  
    Dim locZugriffe As Integer = 1000000  
    Dim locVorlageAdressen As ArrayList  
    Dim locAdressen As New Hashtable  
    Dim locTestKeys(locZugriffsElemente) As String  
    Dim locZeitmesser As New HighSpeedTimeGauge  
    Dim locRandom As New Random(Now.Millisecond)
```

```
    'Warten auf Startschuss.  
    Console.WriteLine("Drücken Sie Return, um zu beginnen", locZeitmesser.DurationInMilliseconds)  
    Console.ReadLine()
```

```
    'Viele Adressen erzeugen:  
    Console.Write("Erzeugen von {0} zufälligen Adresseneinträgen...", locAnzahlAdressen)  
    locZeitmesser.Start()  
    locVorlageAdressen = adresse.ZufallsAdressen(locAnzahlAdressen)  
    locZeitmesser.Stop()  
    Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliseconds)  
    locZeitmesser.Reset()
```

```
    'Aufbauen der Hashtable.  
    Console.Write("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)  
    locZeitmesser.Start()  
    For Each adresse As Adresse In locVorlageAdressen  
        locAdressen.Add(adresse.Matchcode, adresse)  
    Next  
    locZeitmesser.Stop()  
    Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliseconds)  
    locZeitmesser.Reset()
```

```
    '51 zufällige Adressen rauspicken.  
    For i As Integer = 0 To locZugriffsElemente  
        locTestKeys(i) = DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)), _  
            Adresse).Matchcode
```

```

Next

Dim locTemp As Object
Dim locTemp2 As Object

'Zugreifen und Messen, wie lange das dauert,
'das ganze 5 Mal, um die Messung zu bestätigen.
For z As Integer = 1 To locMessungen
    Console.WriteLine()
    Console.WriteLine("{0}. Messung:", z)
    For i As Integer = 0 To locZugriffsElemente
        Console.WriteLine("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
        locTemp = locTestKeys(i)
        locZeitmesser.Start()
        For j As Integer = 1 To locZugriffe
            locTemp2 = locAdressen(locTemp)
        Next j
        locZeitmesser.Stop()
        locTemp = locTemp2.GetType
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliseconds)
    Next

    'Zugriff auf ArrayList für Vergleich
    For i As Integer = 0 To locZugriffsElemente
        Console.WriteLine("{0} Zugriffe auf ArrayList-Element in ", locZugriffe)
        locZeitmesser.Start()
        For j As Integer = 1 To locZugriffe
            locTemp2 = locVorlageAdressen(0)
        Next j
        locZeitmesser.Stop()
        locTemp = locTemp2.GetType
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliseconds)
    Next
Next

Console.ReadLine()
End Sub

```

Wenn Sie das Programm starten, erzeugt es eine Ausgabe, etwa wie im folgenden Bildschirmauszug:

Drücken Sie Return, um zu beginnen

```

Erzeugen von 1000000 zufälligen Adresseneinträgen...fertig nach 3594 ms
Aufbauen der Hashtable mit zufälligen Adresseneinträgen...fertig nach 869 ms

```

```

1. Messung:
1000000 Zugriffe auf: NeAx00563508 in 212 ms
1000000 Zugriffe auf: PlCh00288965 in 213 ms
1000000 Zugriffe auf: VüBe00917935 in 208 ms
1000000 Zugriffe auf ArrayList-Element in 15 ms
1000000 Zugriffe auf ArrayList-Element in 14 ms
1000000 Zugriffe auf ArrayList-Element in 14 ms

```

2. Messung:

1000000 Zugriffe auf: NeAx00563508 in 209 ms
1000000 Zugriffe auf: PlCh00288965 in 214 ms
1000000 Zugriffe auf: VüBe00917935 in 224 ms
1000000 Zugriffe auf ArrayList-Element in 16 ms
1000000 Zugriffe auf ArrayList-Element in 18 ms
1000000 Zugriffe auf ArrayList-Element in 18 ms

3. Messung:

1000000 Zugriffe auf: NeAx00563508 in 209 ms
1000000 Zugriffe auf: PlCh00288965 in 208 ms
1000000 Zugriffe auf: VüBe00917935 in 211 ms
1000000 Zugriffe auf ArrayList-Element in 15 ms
1000000 Zugriffe auf ArrayList-Element in 15 ms
1000000 Zugriffe auf ArrayList-Element in 14 ms

Nach dem Programmstart legt das Programm hier im Beispiel 1.000.000 Testelemente an und baut daraus die Hashtable auf. Anschließend pickt es sich drei Beispieleinträge heraus und misst die Zeit, die es für 1.000.000 Zugriffe auf jeweils diese Elemente der Hashtable benötigt. Das Ganze macht es dreimal, um eine Konstanz in den Verarbeitungsgeschwindigkeiten sicherzustellen. Um im Gegenzug nachzuweisen, wie schnell ein Zugriff auf ein ArrayList-Element erfolgt, führt es eine Messung dazu anschließend durch.

Wieso die Zugriffszeit auf Hashtable-Elemente nahezu konstant ist ...

Sie können die Parameter am Anfang des Programms verändern, um weitere Eindrücke der unglaublichen Geschwindigkeit von .NET zu sammeln. Sie werden bei allen Experimenten jedoch eines herausfinden: Ganz gleich, wie Sie auch an den Schrauben drehen, die Zugriffsgeschwindigkeit auf ein einzelnes Element bleibt nahezu konstant. In den seltensten Fällen benötigt der Zugriff auf ein Element das Doppelte der Durchschnittszeit – und diese Ausreißer sind vergleichsweise selten.

Das Geheimnis für die auf der einen Seite sehr hohe, auf der anderen Seite durchschnittlich gleich bleibende Geschwindigkeit beim Zugriff liegt am Aufbau der Hashtable und an der Behandlung der Schlüssel. Die schnellste Art und Weise, auf ein Element eines Arrays zuzugreifen, ist das direkte Auslesen des Elementes über seinen Index (auch das hat das vorherige Beispiel mit dem Zugriff auf die ArrayList-Elemente gezeigt). Daraus folgt, dass es am günstigsten ist, die Elemente der Hashtable nicht nach einem Schlüssel durchsuchen zu müssen, sondern die Positionsnummer eines Elementes der Hashtable irgendwie zu berechnen. Und genau hier setzt das *Hashing*-Konzept an. *Hashing* bedeutet eigentlich »zerhacken«. Das klingt sehr negativ, doch das Zerhacken des Schlüssels, das auf eine bestimmte Weise tatsächlich stattfindet, dient hier einem konstruktiven Zweck: Wenn der Schlüssel, der beispielsweise in Form einer Zeichenkette vorliegt, *gehashed* wird, geht daraus eine Kennzahl hervor, die Aufschluss über die Wertigkeit der Zeichenkette gibt. *Wertigkeit* in diesem Zusammenhang bedeutet, dass gleiche Zeichenketten nicht nur gleiche Hash-Werte ausweisen, sondern auch, dass größere Zeichenketten größere Hash-Werte bedeuten.

Ein einfaches Beispiel soll diese Zusammenhänge klarstellen: Angenommen, Sie haben 26 Wörter, die alle mit einem anderen Buchstaben beginnen. Diese Wörter liegen in unsortierter Reihenfolge vor. Nehmen wir weiter an, dass Ihr Hash-Algorithmus ganz einfach gestrickt ist: Der Hashcode einer Zeichenfolge entspricht der Nummer des Anfangsbuchstabens jedes Wortes. In dieser vereinfachten Konstellation haben Sie das Problem des Positionsberechnens bereits gelöst. Ihr Hashcode ist die

Indexnummer im Array; sowohl das Einsortieren als auch das spätere Auslesen funktioniert in Windeseile über die Zeichenfolge selbst (genauer über seinen Anfangsbuchstaben).

Das Problem gestaltet sich in der Praxis natürlich nicht ganz so einfach. Mehr Zeichen (um beim Beispiel Zeichenfolgen für Schlüssel zu bleiben) müssen berücksichtigt werden, um den Hash zu berechnen, und bei langen Zeichenketten und begrenzter Hashcode-Präzision kann man nicht ausschließen, dass unterschiedliche Zeichenketten gleiche Hashcodes ergeben. In diesem Fall müssen Kollisionen bei der Indexberechnung berücksichtigt werden. So groß gestaltet sich das Problem aber gar nicht, denn wenn beispielsweise beim Einsortieren der Elemente der sich durch den Hashcode des Schlüssels ergebende Index bereits belegt ist, nimmt man eben den nächsten freien.

Um beim Beispiel zu bleiben: Sie haben nun 26 Elemente, von denen alle mit einem anderen Anfangsbuchstaben beginnen, mit einer Unregelmäßigkeit: Sie haben zwei A-Wörter, ein B-Wort und kein C-Wort. Der Hash-Algorithmus bleibt der gleiche. Sie sortieren das B-Wort in Slot 2, anschließend das erste A-Wort in Slot 1. Nun bekommen Sie das zweite A-Wort zum Einsortieren, und laut Hashcode zeigt es auch auf Slot 1. In diesem Fall fangen Sie an zu suchen, und testen Slot 2, der durch das B-Wort schon belegt ist, aber Sie finden anschließend einen freien Slot 3. Der gehört eigentlich zu C, aber in diesem Fall ist er momentan nicht nur frei, sondern wird auch nie beansprucht werden, da es kein C-Wort in der einzusortierenden Liste gibt. Im ungünstigsten Fall gibt es in diesem Beispiel zwar ein C-Wort, dafür aber kein Z-Wort, und das zweite A-Wort wird als letztes Element einsortiert. Jetzt verläuft die Suche über alle Elemente und landet schließlich auf dem letzten Element für das Z.

... und wieso Sie das wissen sollten!

Sie können sich die Verminderung solcher Fälle mit zusätzlichem Speicherplatz erkaufen. Angenommen, Sie reservieren in unserem Beispiel doppelt so viel Speicher für die Schlüssel, dann sind zwar ganz viele Slots leer, aber das zweite A-Wort muss nicht den ganzen Weg bis zum Z-Index antreten. Was ganz wichtig ist: Sie wissen jetzt, was ein *Load-Faktor* ist. So nennt man nämlich den Faktor, der das Verhältnis zwischen Wahrscheinlichkeiten von Zuordnungskollisionen und benötigtem Speicher angibt. Weniger Kollisionswahrscheinlichkeit erfordert höheren Speicher und umgekehrt. Und Sie können diesen Zusammenhang auch am Beispielprogramm austesten.

WICHTIG: Um dieses Beispiel mit plausiblen Ergebnissen nachvollziehen zu können, benötigen Sie mindestens 1 GByte-Hauptspeicher (unter Windows XP). Anderenfalls beginnt Ihr Computer während des Programmlaufs Speicher auf die Festplatte auszulagern, und dieser Vorgang kann die Messergebnisse natürlich eklatant verfälschen!

Am Anfang des Moduls in der Sub Main des Beispielprogramms finden Sie einen Block mit auskommentierten Deklarationsanweisungen. Tauschen Sie die Auskommentierung der beiden Blöcke, um folgende Parameter für den nächsten Versuch zu Grunde zu legen:

```
Dim locAnzahlAdressen As Integer = 1000000
Dim locZugriffsElemente As Integer = 25
Dim locMessungen As Integer = 3
Dim locZugriffe As Integer = 1000000
Dim locVorlageAdressen As ArrayList
Dim locAdressen As New Hashtable(100000, 1)
Dim locTestKeys(locZugriffsElemente) As String
Dim locZeitmesser As New HighSpeedTimeGauge
Dim locRandom As New Random(Now.Millisecond)
```

Mit diesem Block erhöhen wir die Anzahl der Elemente, die es zu testen gilt, auf 50, und damit erhöhen wir natürlich auch die Wahrscheinlichkeit, Elemente zu finden, die kollidieren. Gleichzeitig verändern wir – im Codelisting fett markiert – den Load-Faktor der Hashtable. Bei der Instanzierung einer Hashtable können Sie, neben der Anfangskapazität, als zweiten Parameter den Load-Faktor bestimmen. Gültig sind dabei Werte zwischen 0.1 und 1. Für den ersten Durchlauf bestimmen wir einen Load-Faktor von 1. Dabei wird auf Speicherplatz auf Kosten von vielen zu erwartenden Kollisionen und damit auf Kosten von Geschwindigkeit verzichtet (dieser Wert entspricht übrigens der Voreinstellung, die dann verwendet wird, wenn Sie keinen Parameter für den Load-Faktor angeben).

Starten Sie das Programm mit diesen Einstellungen, und beenden Sie es zunächst **nicht**, nachdem der Messdurchlauf abgeschlossen wurde! Je nach Leistungsfähigkeit Ihres Rechners sehen Sie bei der Ausgabe der Testergebnisse Messungen, von denen ich nur einige ausschnittsweise im Folgenden zeigen möchte:

```
1000000 Zugriffe auf: HoGa00919471 in 308 ms
1000000 Zugriffe auf: RoMi00603881 in 250 ms
1000000 Zugriffe auf: HeUw00854353 in 302 ms
1000000 Zugriffe auf: LaTh00018037 in 479 ms
1000000 Zugriffe auf: HeGu00415902 in 227 ms
1000000 Zugriffe auf: ViK100627961 in 329 ms
1000000 Zugriffe auf: NeJo00414018 in 232 ms
1000000 Zugriffe auf: JuK100179451 in 344 ms
1000000 Zugriffe auf: VüJo00984374 in 301 ms
1000000 Zugriffe auf: HoUw00216841 in 224 ms
1000000 Zugriffe auf: AlJo00275939 in 249 ms
1000000 Zugriffe auf: AlHa00486261 in 251 ms
1000000 Zugriffe auf: WeKa00572480 in 249 ms
1000000 Zugriffe auf: HeAl00400375 in 229 ms
1000000 Zugriffe auf: SoK100216384 in 298 ms
```

Sie erkennen, dass die Zugriffsdauer auf die Elemente in dieser Liste erheblich streut. Die Zugriffszeit auf einige Elemente liegt teilweise doppelt so hoch wie im Durchschnitt.

Rufen Sie nun, ohne das Programm zu beenden, den Task-Manager Ihres Betriebssystems auf. Dazu klicken Sie mit der rechten Maustaste auf die Task-Leiste (auf einen freien Bereich neben dem Startmenü) und wählen den entsprechenden Menüeintrag aus.

Wenn der Task-Manager-Dialog dargestellt ist, wählen Sie die Registerkarte *Prozesse*. Suchen Sie den Eintrag *HashtableDemo*, und merken Sie sich zunächst den Wert, der dort als Speicherauslastung angegeben wurde (etwa wie in Abbildung 19.3 zu sehen).

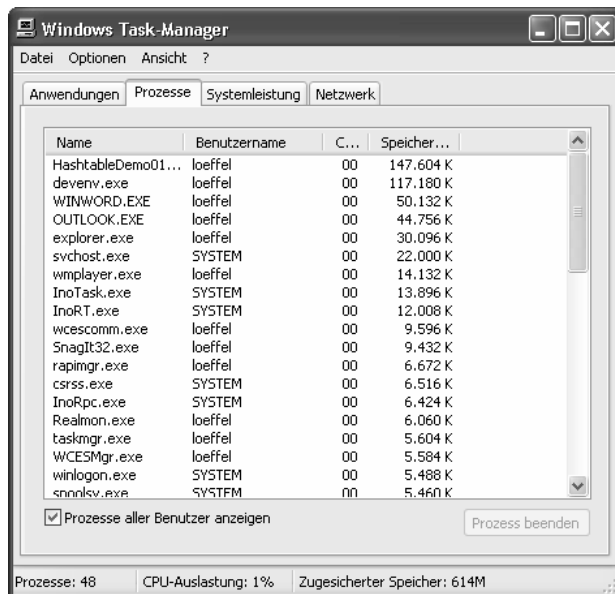


Abbildung 19.3: Bei einem großen Wert für den Parameter Load-Faktor einer Hashtable hält sich der Speicherbedarf in Grenzen, dafür gibt es mehr Zuordnungskollisionen, die Zeit kosten ...

Beenden Sie das Programm anschließend.

Jetzt verändern Sie den Load-Faktor der Hashtable auf den Wert 0.1 und wiederholen die Prozedur:

```
Dim locAdressen As New Hashtable(100000, 0.1)
```

Wenn Sie nicht gerade über mindestens 1.5 GByte Speicher in Ihrem Rechner verfügen, werden Sie den ersten deutlichen Unterschied bereits beim Aufbauen der Hashtable bemerken, das sehr viel mehr Zeit in Anspruch nimmt. Schuld daran ist in diesem Fall auch das Betriebssystem, das Teile des Hauptspeichers zunächst auf die Platte auslagern muss.

Beobachten Sie anschließend, wie sich der kleinere Load-Faktor auf die Zugriffsgeschwindigkeit der Elemente ausgewirkt hat:

```
1000000 Zugriffe auf: MüGa00374433 in 243 ms
1000000 Zugriffe auf: HeBe00168505 in 227 ms
1000000 Zugriffe auf: WeLo00343022 in 225 ms
1000000 Zugriffe auf: ScKa00611039 in 232 ms
1000000 Zugriffe auf: HoDa00523323 in 225 ms
1000000 Zugriffe auf: LöUw00353805 in 231 ms
1000000 Zugriffe auf: WeKa00855227 in 223 ms
1000000 Zugriffe auf: LöUt00696865 in 230 ms
1000000 Zugriffe auf: WeBr00146001 in 226 ms
1000000 Zugriffe auf: JuJü00334704 in 242 ms
1000000 Zugriffe auf: JuKl00583541 in 226 ms
1000000 Zugriffe auf: BrUw00869665 in 229 ms
1000000 Zugriffe auf: ViBr00974013 in 226 ms
1000000 Zugriffe auf: ScMi00635272 in 224 ms
1000000 Zugriffe auf: WeMi00729698 in 230 ms
```

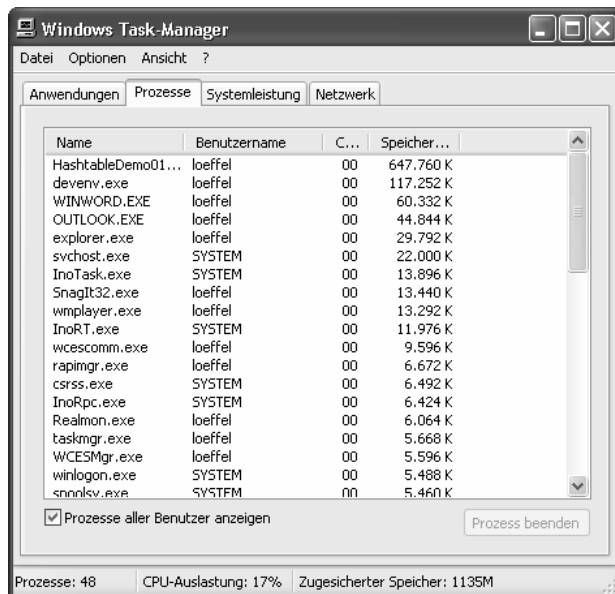


Abbildung 19.4: ... auf der anderen Seite bedeutet ein kleiner Load-Faktor zwar konstantere Zugriffszeiten aber auf Kosten des Speichers

Sie sehen, dass im Gegensatz zum vorherigen Beispiel die Zugriffszeiten nahezu konstant sind. Natürlich kann es bei einer sehr unglücklichen Verteilung immer noch vorkommen, dass es den einen oder anderen Ausreißer gibt. Aber die Anzahl der Ausreißer ist deutlich gesunken.

Betrachten Sie anschließend den Speicherbedarf im Task-Manager, dann sehen Sie sofort, auf wessen Kosten die konstantere Zugriffsgeschwindigkeit erkaufte wurde (Abbildung 19.4).

Der Speicherbedarf der Beispiellapplikation hat sich nahezu verdreifacht!

Es gibt allerdings noch weiteres Beachtenswertes, das Sie im Hinterkopf behalten sollten, wenn Sie mit Hashtable-Objekten programmieren. Diese Dinge zu beachten wird Ihnen jetzt, da Sie wissen, wie Hashtables prinzipiell funktionieren, abermals leichter fallen.

Verwenden eigener Klassen als Schlüssel (Key)

In allen bisherigen Beispielen zur Hashtable wurde eine Zeichenkette als Schlüssel eingesetzt. Der Index auf die Tabelle hat sich dabei aus dem Hashcode der als Schlüssel verwendeten Zeichenkette ergeben.

In unserem konkreten Beispiel ist das Ermitteln des Hashcodes des Strings eigentlich ein zu komplizierter und damit zu lange dauernder Algorithmus. Da wir eine eindeutige »Kundennummer« im Matchcode versteckt haben, können wir rein theoretisch auch diese Nummer als Hashcode verwenden, und der Zugriff auf die Elemente sollte damit – unabhängig vom Load-Faktor – gleich bleibend sein, denn die Kundennummer ist immer eindeutig (im Gegensatz zum Hashcode des Matchcodes, bei dem leicht Dubletten und damit Kollisionen beim Einsortieren in die Hashtable auftreten können).

Wenn Sie Objekte eigener Klassen als Schlüssel verwenden wollen, müssen Sie die beiden folgenden Punkte beherzigen:

- Die Klasse muss ihre Equals-Funktionen überschreiben, damit die Hashtable-Klasse die Gleichheit zweier Schlüssel prüfen kann.
- Die Klasse muss die GetHashCode-Funktion überschreiben, damit die Hashtable-Klasse überhaupt einen *HashCode* ermitteln kann.

WICHTIG: Wenn Sie die Equals-Funktion überschreiben, achten Sie bitte darauf, die richtige Überladungsversion dieser Funktion zu überschreiben. Da die Basisfunktion (die Equals-Funktion von Object) sowohl als nicht statische als auch als statische Funktion implementiert ist, müssen Sie das Overloads-Schlüsselwort anwenden. Da Sie die Funktion überschreiben wollen, müssen Sie Overrides ebenfalls verwenden. Overrides alleine lässt der Compiler nicht zu, da die Funktion schon in die Basisklasse überladen ist. Allerdings – und das ist das Gefährliche – lässt er ein einzelnes Overloads zu, und das führt zu einer Überschattung der Basisfunktion, **ohne** sie zu überschreiben. Das Ergebnis: Der Compiler zeigt Ihnen keine Fehlermeldung (nicht einmal eine Warnung, obwohl er das sollte!), doch Ihre Funktion wird nicht polymorph behandelt und damit nie aufgerufen.

Mit diesem Wissen können wir nun unsere eigene Key-Klasse in Angriff nehmen. Bei unserem stark vereinfachten Beispiel bietet sich ein Teil der Matchcode-Zeichenkette an, direkt als *HashCode* zu fungieren. Da die laufende Nummer einer Adresse Teil des Matchcodes ist, können wir genau diese als *HashCode* verwenden. Der Code für eine eigene Key-Klasse könnte sich folgendermaßen gestalten:

BEGLEITDATEIEN: Sie finden das veränderte Beispielprogramm unter `.\VB 2005 - Entwicklerbuch\E – Datentypen\Kap19\HashtableDemo02` im Verzeichnis der CD zum Buch. Veränderungen in Beispielcode sind im Folgenden fett markiert.

```
Public Class AdressenKey
```

```
    Private myMatchcode As String  
    Private myKeyValue As Integer
```

```
    Sub New(ByVal Matchcode As String)  
        myKeyValue = Integer.Parse(Matchcode.Substring(4))  
        myMatchcode = Matchcode  
    End Sub
```

```
    'Wird benötigt, um bei Kollisionen den richtigen  
    'Schlüssel zu finden.
```

```
    Public Overloads Function Equals(ByVal obj As Object) As Boolean  
        'If Not (TypeOf obj Is AdressenKey) Then  
        '    Dim up As New InvalidCastException("AdressenKey kann nur mit Objekten gleichen Typs verglichen werden")  
        '    Throw up  
        'End If  
        Return myKeyValue.Equals(DirectCast(obj, AdressenKey).KeyValue)  
    End Function
```

```
    'Wird benötigt, um den Index zu "berechnen".
```

```
    Public Overrides Function GetHashCode() As Integer  
        Return myKeyValue  
    End Function
```

```

Public Overrides Function ToString() As String
    Return myKeyValue.ToString
End Function

Public Property KeyValue() As Integer
    Get
        Return myKeyValue
    End Get
    Set(ByVal Value As Integer)
        myKeyValue = Value
    End Set
End Property

```

End Class

Am ursprünglichen Testprogramm selbst sind nur einige kleinere Änderungen notwendig, um die neue Key-Klasse zu implementieren. Die Adresse-Klasse muss dabei überhaupt keine Änderung erfahren. Lediglich am Hauptprogramm müssen einige Zeilen geändert werden, um die Änderung wirksam werden zu lassen.

Module HashtableDemo

```

Sub Main()

    Dim locAnzahlAdressen As Integer = 1000000
    Dim locZugriffsElemente As Integer = 50
    Dim locMessungen As Integer = 5
    Dim locZugriffe As Integer = 1000000
    Dim locVorlageAdressen As ArrayList
    Dim locAdressen As New Hashtable(100000, 0.1)
    Dim locTestKeys(locZugriffsElemente) As AdressenKey
    Dim locZeitmesser As New HighSpeedTimeGauge
    Dim locRandom As New Random(Now.Millisecond)

    .
    . ' Aus Platzgründen weggelassen.
    .
    'Aufbauen der Hashtable
    Console.WriteLine("Aufbauen der Hashtable mit zufälligen Adresseneinträgen...", locAnzahlAdressen)
    locZeitmesser.Start()
    For Each adresse As Adresse In locVorlageAdressen
        'Änderung: Nicht den String, sondern ein Key-Objekt verwenden
        locAdressen.Add(New AdressenKey(adresse.Matchcode), adresse)
    Next
    locZeitmesser.Stop()
    Console.WriteLine("fertig nach {0} ms", locZeitmesser.DurationInMilliseconds)
    locZeitmesser.Reset()

    '51 zufällige Adressen rauspicken.
    'Anderung: Die Keys werden abgespeichert, nicht der Matchcode.
    For i As Integer = 0 To locZugriffsElemente
        locTestKeys(i) = New AdressenKey( _
            DirectCast(locVorlageAdressen(locRandom.Next(locAnzahlAdressen)), Adresse).Matchcode)
    Next

```

```

'Änderung: Kein Object mehr, sondern direkt ein AdressenKey.
Dim locTemp As AdressenKey
Dim locTemp2, locTemp3 As Object

'Zugreifen und messen, wie lange das dauert,
'Das ganze fünfmal, um die Messung zu bestätigen.
For z As Integer = 1 To locMessungen
    Console.WriteLine()
    Console.WriteLine("{0}. Messung:", z)
    For i As Integer = 0 To locZugriffsElemente
        Console.WriteLine("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
        locTemp = locTestKeys(i)
        locZeitmesser.Start()
        For j As Integer = 1 To locZugriffe
            locTemp2 = locAdressen(locTemp)
        Next j
        locZeitmesser.Stop()
        locTemp3 = locTemp2.GetType
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliseconds)
    Next

    'Zugriff auf ArrayList für Vergleich
    .
    . ' Aus Platzgründen ebenfalls weggelassen.
    .
Next
Console.ReadLine()
End Sub

```

Die entscheidende Zeile im Beispielcode hat übrigens gar keine Änderung erfahren müssen. `locTemp` dient nach wie vor als Objektvariable für den Schlüssel, nur ist sie nicht mehr vom Typ `String` definiert, sondern als `AdressenKey`. `locTemp3` (und ehemals `locTemp`) dienen übrigens nur dazu, dass der Compiler die innere Schleife nicht wegoptimiert⁶ und damit Messergebnisse verfälscht.

Wenn Sie dieses Programm starten, werden sie zwei Dinge feststellen. Der Zugriff auf die Daten ist spürbar schneller geworden, und: Der Zugriff auf die Daten erfolgt unabhängig vom Load-Faktor immer gleich schnell. Da der Schlüssel auf die Daten nun eindeutig ist, braucht sich die Hashtable nicht mehr um Kollisionen zu kümmern – es gibt nämlich keine mehr. Folglich bringt die Reservierung zusätzlicher Elemente auch keinen nennenswerten Vorteil mehr. Ganz im Gegenteil: Sie würden Speicher verschwenden, der gar nicht benötigt würde.

In diesem Beispiel sind die zu verwaltenden Datensätze nicht sonderlich groß gewesen. Wenn Sie überlegen, wie viele Zugriffe auf die Objekte der Hashtable tatsächlich notwendig gewesen sind, um überhaupt in einen messbaren Bereich zu gelangen, dann wird deutlich, dass sich der betriebene Aufwand für dieses Beispiel eigentlich nicht gelohnt hat. Dennoch: Denken Sie immer daran, dass Maschinen, auf denen Ihre Software später läuft, in der Regel nicht so leistungsfähig sind wie die Maschinen, auf denen sie entwickelt wird.

⁶ Dieser Handgriff dient nur als Vorsichtsmaßnahme. Ich gebe zu, nicht wirklich überprüft zu haben, ob die Zeile wegoptimiert werden würde. Bei der Intelligenz moderner Compiler (oder JITter) ist das aber stark anzunehmen.

Schlüssel müssen unveränderlich sein!

Wenn Sie sich dazu entschlossen haben, eigene Klassen für die Verwaltung von Schlüsseln in einer Hashtable zu entwickeln, sollten Sie zusätzlich zum Gesagten Folgendes unbedingt beherzigen: Schlüssel müssen unveränderlich sein. Achten Sie darauf, dass Sie den Inhalt eines Key-Objektes nicht von außen verändern, solange er einer Hashtable zugeordnet ist. In diesem Fall würden Sie riskieren, dass die GetHashCode-Funktion unter Umständen einen falschen Hashcode für ein Key-Objekt zurückliefert. Der Nachschlagealgorithmus der Hashtable hätte dann keine Chance mehr, das Objekt in der Datentabelle wieder zu finden.

Enumerieren von Datenelementen in einer Hashtable

Die Enumeration einer *Hashtable* (das Iterieren durch sie mit For/Each) ist prinzipiell möglich. Allerdings müssen sie zwei Dinge dabei beachten:

- Datenelemente werden in einer Hashtable nicht in sequentieller Reihenfolge gespeichert. Da der Hashcode eines zu speichernden Objektes ausschlaggebend für die Position des Objektes innerhalb der Datentabelle ist, kann die wirkliche Position eines Objektes nur bei ganz einfachen Hashcode-Algorithmen vorausgesagt werden. Wenn Sie – und das wird wahrscheinlich am häufigsten der Fall sein – ein String-Objekt als Schlüssel verwenden, wirken die zu speichernden Objekte schon mehr oder weniger zufällig in der Tabelle verteilt.
- Objekte werden innerhalb einer Hashtable in so genannten Bucket-Strukturen gespeichert. Ein Schlüssel gehört untrennbar zu seinem eigentlichen Objekt, und beide werden in einem Bucket-Element in der Tabelle abgelegt. Eine Iteration durch die Datentabelle kann aus diesem Grund nur mit einem speziellen Objekt erfolgen – nämlich vom Typ DictionaryEntry (etwa: *Wörterbucheintrag*).

Eine Iteration durch eine Hashtable könnte für unser Beispiel folgendermaßen aussehen:

```
'Iterieren durch die Hashtable
For Each locDE As DictionaryEntry In locAdressen
    'in unserem Beispiel für den Key
    Dim locAdressenKey As AdressenKey = DirectCast(locDE.Key, AdressenKey)
    'in unserem Beispiel für das Objekt
    Dim locAdresse As Adresse = DirectCast(locDE.Value, Adresse)
Next
```

Typsichere Hashtable

Bei der Entwicklung einer typsicheren Hashtable stehen wir vor ähnlichen Problemen, wie bei einer typsicheren ArrayList. Da ich aus Platzgründen die komplette Theorie nicht wiederholen möchte, gehe ich einfach davon aus, dass Sie den ►Abschnitt »Typsichere Auflistungen auf Basis von CollectionBase« ab Seite 568 durchgearbeitet haben.

Das folgende Beispiel demonstriert den Einsatz einer typsicheren Hashtable an der Erweiterung des vorherigen Beispiels.

BEGLEITDATEIEN: Im Ordner `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\Hashtable03\` finden Sie das veränderte Beispielprogramm im Verzeichnis der CD zum Buch. Veränderungen im Beispielcode sind fett markiert.

Zunächst finden Sie in der Klassendatei *Daten.vb* des Projektes zusätzlich zu der vorhandenen Klasse die Klasse *Adressen*. Sie ist aus der abstrakten Klasse *DictionaryBase* abgeleitet, die die Grundfunktionalität der *Hashtable* beinhaltet:

```
'Typsichere Adressen-Auflistung auf Wörterbuchbasies
Public Class Adressen
    Inherits DictionaryBase

    'Default-Eigenschaft erlaubt das Auslesen der typsicheren Hashtable.
    Default Public Property Item(ByVal key As AdressenKey) As Adresse
        Get
            Return DirectCast(Dictionary(key), Adresse)
        End Get
        Set(ByVal Value As Adresse)
            Dictionary(key) = Value
        End Set
    End Property

    'Liefert eine ICollection aller Keys zurück.
    Public ReadOnly Property Keys() As ICollection
        Get
            Return Dictionary.Keys
        End Get
    End Property

    'Liefert eine ICollection aller Werte zurück.
    Public ReadOnly Property Values() As ICollection
        Get
            Return Dictionary.Values
        End Get
    End Property

    'Erlaubt das Hinzufügen eines Eintrags typsicher.
    Public Sub Add(ByVal key As AdressenKey, ByVal value As Adresse)
        Dictionary.Add(key, value)
    End Sub

    'Überprüft, ob ein bestimmter Key in der Liste enthalten ist.
    Public Function Contains(ByVal key As AdressenKey) As Boolean
        Return Dictionary.Contains(key)
    End Function

    'Entfernt ein Element aus der Liste mithilfe seines Keys.
    Public Sub Remove(ByVal key As AdressenKey)
        Dictionary.Remove(key)
    End Sub
End Class
```

Anstelle der *Hashtable* setzen wir anschließend im Hauptprogramm diese Klasse ein. Die Änderungen dafür sind minimal:

```
Dim locAnzahlAdressen As Integer = 1000000
Dim locZugriffsElemente As Integer = 50
Dim locMessungen As Integer = 5
Dim locZugriffe As Integer = 1000000
Dim locVorlageAdressen As ArrayList
```

```

Dim locAdressen As New Adressen
Dim locTestKeys(locZugriffsElemente) As AdressenKey
Dim locZeitmesser As New HighSpeedTimeGauge
Dim locRandom As New Random(Now.Millisecond)

```

Die erste Änderung betrifft die Deklaration der Hashtable am Anfang des Programms (siehe fett markierte Zeile). Sie wird hier nicht mehr als Hashtable-Objekt, sondern als Objekt vom Typ Adressen deklariert – damit wird die verwendete Hashtable typsicher.

```

Dim locTemp As AdressenKey
'Typsichere Hashtable: Indexer liefert direkt Adresse-Objekt zurück
Dim locTemp2 As Adresse
Dim locTemp3 As Object

'Zugreifen und Messen, wie lange das dauert,
'Das ganze fünfmal, um die Messung zu bestätigen.
For z As Integer = 1 To locMessungen
    Console.WriteLine()
    Console.WriteLine("{0}. Messung:", z)
    For i As Integer = 0 To locZugriffsElemente
        Console.WriteLine("{0} Zugriffe auf: {1} in ", locZugriffe, locTestKeys(i))
        locTemp = locTestKeys(i)
        locZeitmesser.Start()
        For j As Integer = 1 To locZugriffe
            locTemp2 = locAdressen(locTemp)
        Next j
        locZeitmesser.Stop()
        locTemp3 = locTemp2.GetType
        Console.WriteLine("{0} ms", locZeitmesser.DurationInMilliseconds)
    Next
    .
    . ' Aus Platzgründen ausgelassen
    .
Next
Console.ReadLine()

```

Die zweite Änderung betrifft den Rückgabewerttyp, der zurückgeliefert wird, wenn die typsichere Hashtable per Index ausgelesen wird. locTemp2 ist nunmehr direkt vom Typ Adresse definiert, und die Zuweisung aus einem Element der Hashtable läuft – dank Typsicherheit – ohne Casting ab.

HINWEIS: Wie Sie selbst anhand der ermittelten Zeiten erkennen können, geht der Einbau der Typsicherheit auf Kosten der Geschwindigkeit. Anders ist das, wenn Sie generische Wörterbuchauflistungsklassen wie beispielsweise KeyedCollection verwenden, über die Sie in ► Kapitel 20 genauere Ausführungen finden.

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 `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\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()  
  
    '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 `.|VB 2005 - Entwicklerbuch|E - Datentypen|Kap19|CollectionsDemo` finden. Verändern Sie das Programm, sodass 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 `.\VB 2005 - Entwicklerbuch\E - Datentypen\Kap19\SortedListDemo`. Es besteht aus zwei 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.

Zugriff auf Elemente der SortedList per Indexer

Wenn Sie per Index auf die Elemente der SortedList zugreifen, verwenden Sie dazu deren GetByIndex-Eigenschaft, so wie in der fett markierten Zeile im Listing zu sehen. Möchten Sie ein Element in der SortedList verändern und dabei nicht einen Schlüssel zur Positionsbestimmung, sondern seinen Index verwenden, verwenden Sie die Methode SetByIndex.

HINWEIS: Wenn Sie eine bestimmte Synchronisation zwischen Ihrem Schlüssel und dem eigentlichen Objekt einhalten müssen, sollten Sie diese letzte Methode allerdings nicht verwenden. Da Sie in diesem Fall nur das eigentliche Datenobjekt in der Liste, nicht aber dessen Schlüssel verändern, könnte unter Umständen Ihre eigene Datenstruktur bei der Datenverwaltung verletzt werden.

Sortierung einer SortedList nach beliebigen Datenkriterien

Beim Hinzufügen eines Schlüssel-/Wertepaares zu einem SortedList-Objekt ist dessen Schlüssel ausschlaggebend für die Sortierreihenfolge – es ist eigentlich nicht möglich, die Liste nach bestimmten Eigenschaften des hinzugefügten Objektes sortieren zu lassen, auch nicht mit speziellen IComparer-eingebundenen Klassen. Da doppelte Werte innerhalb der von SortedList verwalteten Liste vorkommen können, können Sie auch nicht jede beliebige Eigenschaft des einzusortierenden Objek-

tes als Schlüssel verwenden, da die Schlüssel eindeutig sein müssen. Dummerweise ist das genau der Knackpunkt, denn: CompareTo der jeweils verwendeten Key-Klasse wird ja ebenfalls dazu verwendet, damit SortedList herausfinden kann, ob es bereits einen Schlüssel in der Liste gibt.

Allerdings: Sie können den folgenden Trick anwenden, um eine SortedList dennoch nach beliebigen Eigenschaften der zu speichernden Klasse zu sortieren, auch wenn das erheblich auf Kosten der Performance geht!

WARNUNG: Deswegen die dringende Warnung an dieser Stelle: Mit dem folgenden Kniff tricksen wir den Algorithmus zur Platzierung der Elemente einer SortedList komplett aus – und zwar auf die Kosten seiner Effektivität. Wenden Sie diesen Algorithmus nur an, wenn Sie wenige Elemente sortieren müssen.

Schauen Sie sich den folgenden Code der Klasse AdressenKey an, deren instanzierte Objekte als Schlüssel der Daten fungieren sollen:

```
Public Class AdressenKey
    Implements IComparable

    Private myMatchcode As String
    Private myKeyValue As Integer
    Private myDataToSort As String

    Sub New(ByVal Matchcode As String, ByVal DataToSort As String)
        myKeyValue = Integer.Parse(Matchcode.Substring(0, 8))
        myMatchcode = Matchcode
        myDataToSort = DataToSort
    End Sub

    'Wird benötigt, um bei Kollisionen den richtigen
    'Schlüssel zu finden.
    Public Overloads Overrides Function Equals(ByVal obj As Object) As Boolean
        Return myKeyValue.Equals(DirectCast(obj, AdressenKey).KeyValue)
    End Function

    'Wird benötigt, um den Index zu "berechnen".
    Public Overrides Function GetHashCode() As Integer
        Return myKeyValue
    End Function

    Public Overrides Function ToString() As String
        Return myKeyValue.ToString
    End Function

    Public Property KeyValue() As Integer
        Get
            Return myKeyValue
        End Get
        Set(ByVal Value As Integer)
            myKeyValue = Value
        End Set
    End Property
End Class
```

```

Public Property DataToSort() As String
    Get
        Return myDataToSort
    End Get
    Set(ByVal Value As String)
        myDataToSort = Value
    End Set
End Property

Public Function CompareTo(ByVal obj As Object) As Integer Implements System.IComparable.CompareTo
    If myMatchcode = DirectCast(obj, AdressenKey).myMatchcode Then
        Return 0
    End If
    If DataToSort.CompareTo(DirectCast(obj, AdressenKey).DataToSort) = 0 Then
        Return -1
    Else
        Return DataToSort.CompareTo(DirectCast(obj, AdressenKey).DataToSort)
    End If
End Function
End Class

```

Sie stellen fest, dass der Konstruktor der Klasse nicht nur den eigentlichen als Schlüssel fungierenden Wert, sondern ein weiteres Datenfeld aufnimmt. Dieses weitere Datenfeld ist – um das Beispiel einfach zu halten – ein String, könnte rein theoretisch aber auch jedes andere Objekt sein.

Der Trick funktioniert nun folgendermaßen: Die CompareTo-Methode muss primär so funktionieren, dass doppelte Schlüssel ermittelt und durch eine Ausnahme, die SortedList im Bedarfsfall erzeugt, ausgeschlossen werden können. CompareTo liefert also dann 0 zurück, wenn es sich bei den zu vergleichenden AdressenKey-Instanzen um gleiche Schlüssel handelt. Wenn das nicht der Fall ist, werden allerdings nicht die wirklichen Schlüssel der Größe nach verglichen, sondern die Sortierfelder. Natürlich können diese ebenfalls gleich sein, doch sind dieses Mal Dubletten gestattet. SortedList darf davon aber nichts mitbekommen, weil SortedList sonst wiederum von der Gleichheit der Schlüssel ausgehen würde – eine Ausnahme wäre die Folge. Also liefern wir, selbst wenn die beiden Sortierfelder gleich sind, den Wert für *kleiner* als Ergebnis zurück. Da es sowieso egal ist, wenn beispielsweise ausschließlich nach Nachnamen sortiert wird, welcher der doppelten »Thiemänner« an erster Stelle kommt, hat das Zurückliefern des »falschen« Vergleichsergebnisses in der Praxis keine Auswirkungen.

Diesen Trick sehen Sie im oben gezeigten Listing in den fettgedruckten Zeilen umgesetzt. Der Wert 0 für *gleich* wird nur bei gleichen Matchcodes (den Schlüsseln) zurückgeliefert. Gleiche Matchcodes sind also auch in unserer Auflistung nicht gestattet, damit eine eindeutige Auflösung von Matchcodes zu eigentlichen Datensätzen gewährleistet bleibt. Wenn die Matchcodes ungleich gewesen sind, liefert der nächste Teil der CompareTo-Methode aber das Vergleichsergebnis der Werte zurück, nach denen sortiert werden soll – es sei denn, sie wären gleich. Sind sie es, wird gemogelterweise der Wert *-1* für *kleiner* zurückgeliefert, und dass diese kleine Mogelei in der Praxis keine Auswirkungen hat, zeigt das Testprogramm in Form von Sub SortedByFieldDemo, das Sie laufen lassen können, wenn Sie die Auskommentierung der beiden Zeilen am Anfang des Moduls zurücknehmen:

```

Sub SortedByFieldDemo()
    Dim locZufallsAdressen As ArrayList = Adresse.ZufallsAdressen(20)
    Dim locAdressen As New SortedList
    Dim locAdressenKey As AdressenKey

```

```

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

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

```

Wenn Sie dieses Programm laufen lassen, sehen Sie in etwa folgende Zeilen im Konsolenfenster:

```

Ursprungsanordnung:
00000014WeUt: Weichelt, Uta, 18364 Bielefeld
00000013ThKa: Thiemann, Katrin, 80995 Berlin
00000012JuDa: Jungemann, Daja, 31318 Bad Waldliesborn
00000011TiMa: Tinoco, Margarete, 67423 Rheda
00000010EnCh: Englisch, Christian, 28395 Lippstadt
00000009MeJo: Meier, José, 48230 Soest
00000008SoUt: Sonntag, Uta, 52796 Straubing
00000007MeAn: Meier, Anne, 92606 Rheda
00000006WeJü: Westermann, Jürgen, 76188 Lippetal
00000005TiDa: Tinoco, Daja, 46492 Rheda
00000004SoBr: Sonntag, Britta, 89217 Dortmund
00000003LöUt: Löffelmann, Uta, 93934 Bad Waldliesborn
00000002RoAn: Rode, Anne, 05647 München
00000001AlMe: Albrecht, Melanie, 45944 Wiesbaden
00000000HöLo: Hörstmann, Lothar, 29607 Straubing

Zugriff per Index:
00000001AlMe: Albrecht, Melanie, 45944 Wiesbaden
00000010EnCh: Englisch, Christian, 28395 Lippstadt
00000000HöLo: Hörstmann, Lothar, 29607 Straubing
00000012JuDa: Jungemann, Daja, 31318 Bad Waldliesborn
00000003LöUt: Löffelmann, Uta, 93934 Bad Waldliesborn
00000009MeJo: Meier, José, 48230 Soest
00000007MeAn: Meier, Anne, 92606 Rheda
00000002RoAn: Rode, Anne, 05647 München
00000008SoUt: Sonntag, Uta, 52796 Straubing
00000004SoBr: Sonntag, Britta, 89217 Dortmund
00000013ThKa: Thiemann, Katrin, 80995 Berlin
00000011TiMa: Tinoco, Margarete, 67423 Rheda
00000005TiDa: Tinoco, Daja, 46492 Rheda
00000014WeUt: Weichelt, Uta, 18364 Bielefeld
00000006WeJü: Westermann, Jürgen, 76188 Lippetal

```