

4 Auflistungen

In den meisten Sprachen stellt das Array den einfachsten und häufigsten Typ für strukturierte Daten dar. Im Vergleich zum „alten“ Visual Basic werden Array in .Net vollkommen anders behandelt. Wie bekannt, ist ein Array ein Verweistyp, der über einen Index zugänglich ist. Die Array-Elemente sind also immer über eine Ganzzahl zugänglich. Da Arrays Verweistypen sind, werden diese nicht auf dem Stack sondern auf dem Heap gespeichert.

Auflistungen sind dagegen Klassen, die einen gesteuerten Zugriff auf Objekte oder Variablen eines bestimmten Typs bilden. Auflistungen finden wir an vielen Stellen. Auch die Toolbox enthält verschiedene visuelle Auflistungen wie z.B. *ListView*, *TreeView* über die *Node*-Auflistung. Weiterhin stelle VB.Net eine Vielzahl von Auflistungsklassen zur Verfügung. Ein Array ist im Grunde die einfachste Form einer Auflistung. Eine Auflistung ist im Grunde eine Gruppe oder Sammlung von Objekten.

4.1 Array (Datenfelder)

Ein Array fasst mehrere Variablen des gleichen Typs zu einer Einheit zusammen. Das Array erhält einen gemeinsamen Namen und die einzelnen Komponenten werden über einen Index angesprochen. Der Index beginnt (ab VB.NET) immer mit dem Wert 0. Der bei der Vereinbarung des Arrays in Klammern angegebene Wert stellt den größten erlaubten Index dar. Durch die Vereinbarung

```
Dim aNum As Integer
```

wird also ein Integer-Array mit **5** Elementen (0 ist bereits das erste Element) definiert, die erlaubten Indices sind 0 bis 4.

Die Array-Größe wird erst zur Laufzeit bestimmt und muss daher keine Konstante sein. Somit ist auch die folgende Vereinbarung möglich

```
Dim aSize As Integer = Console.ReadLine()  
Dim arr(aSize) As Integer
```

Die Elemente des erzeugten Arrays werden implizit mit 0 vorbesetzt. Ein bereits gefülltes Array kann wieder mit der Methode *.Clear()* geleert werden.

Beim lesenden oder schreibendem Zugriff auf eine Komponente wird ein geklammerter Indexausdruck verwendet

```
a = aString(0)      ' Lesender Zugriff  
aString = a        ' Schreibender Zugriff
```

Liegt der angegebene Indexwert außerhalb des erlaubten Bereichs, dann wird ein entsprechender Laufzeitfehler ausgelöst; Die *IndexOutOfRangeException*.

```
a = aString(5) ' Lesender Zugriff
```

Ein Array gehört in VB.NET zu den sog. Referenztypen. Bei einem Referenztyp werden die Daten nicht in der Variablen selbst gespeichert. Vielmehr wird der Speicherplatz für die Arrayelemente bei der Objekterzeugung dynamisch auf dem Heap reserviert. In der Array-Variablen selbst wird nur die Startadresse des reservierten Bereichs als Referenz gespeichert.

Ein Array "kennt" seine Größe. Sie kann mit der *Length*-Eigenschaft abgefragt, aber nicht verändert werden

```
nSize = aString.Length
```

Initialisierung eines Array

Ein Array kann schon bei der Vereinbarung mit Werten initialisiert werden. In diesem Fall darf die Größe des Array nicht angegeben werden, das dies als der Initialisierung abgeleitet werden kann.

```
' Array mit 2 Elementen  
Dim aString() As String = {"Element1", "Element2"}
```

Verändern der Größe eines Arrays

Die Größe eines Arrays kann nachträglich verändert werden, jedoch nicht die Anzahl der Dimensionen und nicht der Typ der Elemente.

```
' Größe eines Array verändern  
' Die Werte gehen aber verloren  
ReDim aString(2) As String  
  
' und ist gleichbedeutend mit  
Dim aString(2) As String
```

Sollen die bisher gespeicherten Werte erhalten bleiben, dann ist dies mit der Anweisung *Preserve* möglich. Auch in diesem Fall wird tatsächlich ein neues Array-Objekt der gewünschten Größe erzeugt, aber in einem zusätzlichen Schritt mit den Werten des alten Array-Objekts initialisiert. Die neuen Elemente des Array werden immer mit 0 bei numerischen Werten und Null ("") bei einem String vorbelegt.

```
' Array mit 2 Elementen vergrößern  
ReDim Preserve aString(4)
```

Folgende Anweisung löschen den Inhalt eines Arrays, und lassen keinen Zugriff auf das Array mehr zu. Der vom Array-Objekt belegte Speicherplatz wird automatisch freigegeben.

```
' Die Anweisung  
Erase aString  
  
' oder  
  
aString = Nothing  
  
' sind gleichbedeutend.
```

Array-Zuweisung und Clone eines Array erstellen

Die Zuweisung einer Array-Variablen an eine weitere Array-Variable ist möglich, wenn der Datentyp der Elemente übereinstimmt. Die beiden Arrays können allerdings unterschiedliche Größe haben:

```
' Zuweisung eines Array  
Dim aString1() As String = {"Element1", "Element2"}  
Dim aString2() As String = {"Element1", "Element2", "Element3"}  
  
aString1 = aString2
```

wird jedoch nicht etwa der Inhalt des Arrays kopiert, sondern - wie es bei Referenztypen üblich ist - die in der Referenzvariablen *aString2* gespeicherte Adresse nach *aString1* kopiert. Anschließend sprechen *aString1* und *aString2* dasselbe Array-Objekt an.

Bei der Anweisung

```
'  
aString1(0) = ""
```

wird auch in das Array *aString2(0)* ein Null-String gespeichert. Es wurde also nur eine Referenz auf das zweite Array *aString2* gebildet. Eine Änderung eines Elementes des einen Array hat auch eine Änderung des Elementes im zweiten Array zufolge.

Soll nun eine echte Kopie eines Array erzeugt werden, so muss dies mit der *.Clone()* Anweisung erfolgen.

```
' Kopieren (Clonen) eines Array
Dim aString1() As String = {"Element1","Element2"}
Dim aString2() As String

aString2 = aString1.Clone()
```

Bei der Parameterübergabe eines Arrays liegen ähnliche Verhältnisse vor wie bei einer Zuweisung. Es wird nicht etwa das Array-Objekt selbst, sondern die beim Aufruf angegebene Referenz an die Funktion übergeben. Als Konsequenz daraus sind Modifikationen, die die aufgerufene Funktion am Array vornimmt, auch nach der Rückkehr für den Aufrufer sichtbar.

Mehrdimensionales Array

Mehrdimensionale Arrays, sind Arrays die wie eine Matrix aufgebaut sind. Mit der Anweisung

```
' Anweisung für ein mehrdimensionales Array
Dim aString1(1,2) As Integer
aString1(0,0) = 0
aString1(0,1) = 1
aString1(0,2) = 2
aString1(1,0) = 10
aString1(1,1) = 11
aString1(1,2) = 12
```

wird ein leeres (ohne Werte) 2-dimensionales Array vereinbart, und kann sich wie folgt vorgestellt werden.

(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)

Der Zugriff auf ein Array-Element erfolgt z.B. in der Form

```
aString1(0,0) = 0
```

Ein bekannter Fall für einen mehrdimensionales Array ist das Einlesen einer Verzeichnis Struktur.

```
Dim aSubDir() As String  
aSubdir = Directory.GetFiles("c:\winnt", "*.Bmp")
```

Über eine For-Next Schleife kann nun auf die einzelne Elemente zugegriffen werden.

4.2 ArrayList()

Rezept	13.	Auflistungen
Beispiel		ArrayList

Auch wenn der Name *ArrayList* ein Array vermuten lässt, gehören *ArrayList* zur Klasse *System.Collection*. Einer *ArrayList* können Sie wie bei einem Array Elemente hinzufügen und löschen. Ein Unterschied besteht jedoch darin, dass für die Elemente kein Index angegeben werden muss. Auflistungen werden dynamisch vergrößert, oder verkleinert wenn Elemente eingefügt bzw. gelöscht werden. Die Anzahl von Elementen müssen bei einer *ArrayList* nicht mit *ReDim* oder *ReDim Preserve* angepasst werden, wenn sich die Anzahl von Elementen ändert.

```
Dim alWochentag As New ArrayList()

With alWochentag
    .Add("Montag")
    .Add("Dienstag")
    .Add("Mittwoch")
    .Add("Donnerstag")
    .Add("Freitag")
    .Add("Samstag")
    .Add("Sonntag")
End With
```

Mit den Eigenschaft *.Count()* kann nun die tatsächliche Anzahl der Elemente, mit der Eigenschaft *.Capacity()* wird ermittelt, wie viele Elemente im *ArrayList* Platz haben. Erreicht *.Count()* die Grenze von *.Capacity()* wird automatisch neuer Speicherplatz für die *ArrayList* reserviert. Der Vorbelegte Wert für *.Capacity()* beträgt 16 und wird von der CLR vorgegeben.

```
MessageBox.Show("Anzahl Element:" & alWochentag.Count() & _
    "ArrayListgröße :" & alWochentag.Capacity())
```

Um den durch die automatische Reservierung entstehenden Speicheroverhead bei Bedarf zu reduzieren, kann die *TrimToSize()* – Methode verwendet werden.

```
' Speicheroverhead auf die tatsächlich benötigte Größe reduzieren

alWochentag.TrimToSize()
MessageBox.Show("Anzahl Element:" & alWochentag.Count() & _
    "ArrayListgröße :" & alWochentag.Capacity())
```

Zum Löschen aller Elemente einer ArrayList wird die Methode `.Clear()` aufgerufen. Das Löschen der Elemente führt dazu, dass der Standardwert des **ArrayList** wieder auf 16 gesetzt wird.

```
' ArrayList löschen
alWochentag.Clear()
alWochentag.TrimToSize()
MessageBox.Show("Anzahl Elemente:" & alWochentag.Count() & _
               "ArrayListgröße :" & alWochentag.Capacity())
```

Durchlaufen einer ArrayList

Wollen Sie ein *ArrayList* durchlaufen, verwenden Sie hierzu einfach die *For-Each* Anweisung.

```
Dim alWochentag As New ArrayList()
Dim cTage As String
Dim cTemp As String

' ArrayList füllen

For Each cTage In alWochentag
    cTemp += cTage & CtrlChr.CrLf
Next

MessageBox.Show(cTemp, "ArrayList")
```

Sortieren und Suchen in einer ArrayList

ArrayList kann sortiert werden. Hier zur Verfügung gestellt wird die Methode `.Sort()` zur Verfügung gestellt. Dahinter steht die Standard-Sortierung (aufsteigend) mit dem Interface `IComparer` mit seiner Methode `.Compare()`. Diese Methode kann wie viele andere auch mit einer eigenen Methode ersetzt werden.

```
' ArrayList sortieren; Standard-Methode
alWochentag.Sort()
```

```
' ArrayList sortieren; eigene Methode
Dim myComparer As New DescSort
alWochentag.Sort(myComparer)
```

Eigene Sortierklasse

```
Public Class DescSort
    Implements IComparer
    Public Function Compare(ByVal x As Object, ByVal y As Object) As
        Integer Implements IComparer.Compare
        '-----
        ' Absteigend Sortieren
        '-----
        Return New CaseInsensitiveComparer().Compare(y, x)
    End Function
End Class
```

Suchen in einer ArrayList

```
' ArrayList sortieren
alWochentag.Sort()
Dim oSuche As Object = "Mittwoch"
FindObject(alWochentag, oSuche)
```

Die selbstdefinierte Methode FindObject() dient zum Suchen von Elemente in einer ArrayList. Vor dem Suchen muss die .Sort() Anweisung ausgeführt werden.

```
Public Sub FindObject(ByVal myList As ArrayList, ByVal myObject As Object)
    '-----
    Dim myIndex As Integer = myList.BinarySearch(myObject)

    If myIndex < 0 Then
        Console.WriteLine("Objekt ({0}) nicht gefunden. ", _
            myObject, Not myIndex)
    Else
        Console.WriteLine("Objekt gefunden in Index ({0}) " + "{1}.", _
            myObject, myIndex)
    End If
End Sub
```

4.3 Collection()

Eine Auflistung (Collection) fasst ähnlich wie ein Array mehrere Elemente zu einer Einheit zusammen, dabei werden jedoch Operationen wie das Einfügen und Entfernen von Elementen effizienter durchgeführt als bei einem Array. Es kommt anfangs häufig zu Schwierigkeiten bei der Zuordnung des Begriffs "Collection". Der Name wird einmal als Sammelbegriff für verschiedene Auflistungsklassen verwendet wird, die von der .NET Framework Class Library im Namespace System.Collections zur Verfügung gestellt werden. Visual Basic stellt aber (seit vielen Versionen) eine eigene Klasse mit dem Namen *Collection* zur Verfügung, seit der Version .NET im Namespace Microsoft.VisualBasic.

Auf die Elemente einer Auflistung kann man - je nach Klasse - entweder über einen ganzzahligen Index oder über einen sog. Schlüssel (vom Typ *String*) zugreifen, die Klasse *Collection* unterstützt sogar beide Methoden. Im Folgenden wird der Einsatz der Klasse *Collection* ausführlicher diskutiert. Die Vereinbarung

Hinzufügen von Elementen in ein *Collection*-Objekt.

```
Dim colMyCollection As New Collection
```

Erstellt eine Referenz auf ein neues *Collection*-Objekt.

Danach können diesem *Collection*-Objekt mit der Methode *.Add()* Elemente hinzugefügt werden

```
colMyCollection.Add("Element 1")  
colMyCollection.Add("Element 2")  
colMyCollection.Add(4711)  
colMyCollection.Add(2.7)
```

Da der Übergabeparameter von *.Add()* den Typ *Object* hat, kann beim Aufruf ein beliebiger Datentyp angegeben werden. Die im *Collection*-Objekt enthaltenen Elemente müssen also nicht den gleichen Datentyp haben.

Zugreifen und Auslesen eines *Collection*-Objekt.

Auf die Elemente der *Collection* kann man über einen (aus Gründen der Kompatibilität) 1-basierten Index zugreifen:

```
For i = 1 To colMyCollection.Count
    Console.WriteLine(colMyCollection(i))
Next i
```

Der Ausdruck *colMyCollection(i)* sieht zwar aus wie der Zugriff auf eine Array-Komponente, in Wirklichkeit erfolgt der Zugriff aber über eine indizierte Eigenschaft mit dem Namen *Item*, der Ausdruck ist äquivalent zu. Da es sich bei der *Item*-Eigenschaft um die *Default*-Eigenschaft der Klasse *Collection* handelt, kann ihr Name weggelassen werden.

Mit einer *For Each*-Schleife ist es auch dann möglich, über alle Elemente zu iterieren, wenn die eingesetzte Auflistungs-Klasse keinen Index-Zugriff unterstützt:

```
Dim colElement As Object

For Each colElement In colMyCollection
    Console.Write(colElement & " ")
Next colElement
```

Einige Auflistungs-Klassen (wie auch die Klasse *Collection*) unterstützen den Zugriff auf ein Element über einen eindeutigen Schlüssel des Typs *String*. Der Schlüssel wird als Argument angegeben, wenn das Element der Auflistung hinzugefügt wird:

```
colMyCollection.Add("Element 1", "ELEMENT1")
colMyCollection.Add("Element 2", "ELEMENT2")
colMyCollection.Add(4711, "ELEMENT3")
colMyCollection.Add(2.7, "ELEMENT4")
```

Der Ausdruck *colMyCollection("ELEMENT4")* oder ausführlicher *colMyCollection.Item("ELEMENT4")* liefert dann den Wert 2.7 zurück

Der Schlüssel bleibt im Gegensatz zum Index auch nach dynamischen Veränderungen des *Collection*-Objekts unverändert erhalten

4.4 Hashtabel()

Rezept	14.	Auflistungen
Beispiel		Hashtable

Mit dem .NET Framework kommen etliche neue Collection Klassen die unter dem *System.Collection* Namespace angesprochen werden können. Unter anderem mit dabei ist die *Hashtable* Klasse, die einige neue Funktionalitäten mitbringt. Die *Hashtable* Klasse hat eine gewisse Ähnlichkeit mit der *Collection*-Klasse.

Die Hashtable Klasse wird immer dann zum Einsatz kommen, wenn man auf Elemente per Schlüssel und nicht per Index zugreifen muss - und das bare möglichst schnell

Hinzufügen von Elementen in ein Hashtable-Objekt.

```
Dim hsMyHashtable As New Hashtable
```

Erstellt eine Referenz auf ein neues *Hashtable* -Objekt.

Danach können diesem *Hashtable* -Objekt mit der Methode *.Add()* Elemente hinzugefügt werden

```
hsMyHashtable.Add("Montag", "1")
hsMyHashtable.Add("Dienstag", "2")
hsMyHashtable.Add("Mittwoch", "3")
hsMyHashtable.Add("Donnerstag", "4")
hsMyHashtable.Add("Freitag", "5")
hsMyHashtable.Add("Samstag", "6")
hsMyHashtable.Add("Sonntag", "7")
hsMyHashtable.Add(8, 8)
```

Das Hashtable-Objekt wird immer mit Schlüssel/Wert Paaren aufzufüllen. Im Beispiel werden die verschiedenen Werte Paare vom Typ *String* und *Integer* gespeichert. Allerdings nehmen alle Funktionen der Hashtable Klasse den Typ *Object* entgegen. Somit können verschiedene Datentypen gespeichert werden.

Auslesen eines Hashtable-Objekt.

```
Dim hsMyHashtable As New Hashtable
Dim deInhalt As DictionaryEntry

' Hashtable füllen
...

For Each deInhalt In hsMyHashtable
    Console.WriteLine("Key:" & CType(deInhalt.Key, String) & _
        " Value:" & CType(deInhalt.Value, String))
Next
```

Das Auslesen funktioniert über das *ICollection* Interface der Hashtable Klasse. Für jeden Eintrag bekomme wir eine *DictionaryEntry* Struktur geliefert, die zwei Eigenschaften bereitstellt: *Key* und *Value*.

Suchen in einem Hashtable-Objekt.

Das Besondere an Dictionaries bzw der Hashtable Klasse ist ja der, dass die Schlüssel durch einen Hash repräsentiert werden, der extrem schnell auffindbar ist.

```
Dim hsMyHashtable As New Hashtable
Dim deInhalt As DictionaryEntry

' Hashtable füllen
...

Dim nAnzahl As Integer          ' Anzahl der Elemente ermitteln
nAnzahl = hsMyHashtable.Count

Dim bKeyCont As Boolean        ' Prüfen, ob der Key vorhanden ist
bKeyCont = hsMyHashtable.ContainsKey("Montag")

Dim bValueCont As Boolean      ' Prüfen, ob der Wert vorhanden ist
bValueCont = hsMyHashtable.ContainsValue("1")

If bKeyCont = True Then        ' Anzeigen, wenn gefunden
    Console.WriteLine("Wert:" & CType(hsMyHashtable.Item("Montag"), String))
End If
```

Mit Hilfe der *Count* Eigenschaft kann die Anzahl der gespeicherten Elemente ermittelt werden. Die Eigenschaft *ContainsKey* liefert einen boolschen Rückgabewert, der angibt, ob ein bestimmter Schlüssel existiert. Diese Funktion sollte man unbedingt immer dann aufrufen, wenn man ein Element auslesen, löschen oder updaten möchte. Sonst droht eine Exception, falls das Element nicht vorhanden sein sollte.

Auch die Eigenschaft *ContainsValue* kann sehr nützlich sein. Damit kann man nach einem Wert suchen lassen, und anhand des Rückgabewerts entscheiden, ob ein Element schon in der Hashtable gespeichert ist oder nicht.

Änderungen und Löschen in einem Hashtable-Objekt.

Wie eine einer Collection oder ArrayList können auch im einem Hashtable Objekt Elemente geändert oder gelöscht werden.

```
Dim hsMyHashtable As New Hashtable

... ' Hashtable mit Werte füllen

bKeyCont = hsMyHashtable.ContainsKey(8) ' Key Suchen
If bKeyCont = True Then
    hsMyHashtable.Item(8) = 88
End If

' Element im Hashtable löschen
If bKeyCont = True Then
    hsMyHashtable.Remove(88)
End If

bKeyCont = hsMyHashtable.ContainsKey(88)
If bKeyCont = False Then
    Console.WriteLine("Element nicht gefunden")
End If

' Gesamte Hashtable löschen
hsMyHashtable.Clear()
If hsMyHashtable.Count = 0 Then
    Console.WriteLine("Hashtable gelöscht")
End If
```

Die Eigenschaft *ContainsKey* ist bereits bekannt, ebenso wie der Indexer - nur dass dieser hier zum Ändern eines Wertes verwendet wird. Ebenso einfach ist die Verwendung der *Remove*-Methode, die ein einzelnes Element anhand des Schlüssels aus dem Hashtable entfernt. Mit die *Clear*-Methode werden nun alle Elemente aus dem Hashtable wider gelöscht.

Hinweis:

Vor jedem Anzeigen, Löschen oder Änderung von einem Key oder Wert erst mal prüfen ob der gesuchte Key oder Wert auch wirklich vorhanden ist, da sonst eine Exception ausgelöst wird.

4.5 Queue() oder auch Warteschlange

Rezept	15.	Auflistungen
Beispiel		Queue

Eine Queue() oder auch Warteschlange ist eine Struktur nach dem FiFo Prinzip (Firt-In/First-Out). Die Funktion kann mit einer Warteschlange am einem Kiosk verglichen werden. In eine Warteschlange stehen viele Personen, aber immer nur die erste Person kann bedient werden. In einer Warteschlange werden Informationen nach dem Eingang sequenziell gespeichert und abgearbeitet. Wie bei einem ArrayList wird die Kapazität automatisch angepasst, wenn die maximale Anzahl von Elementen erreicht wurde. Ähnlich wie bei anderen Auflistungen können Elemente in die Warteschlange aufgenommen werden. Bei der *Queue()*-Klasse wird dies mit der Methode *.Enqueue()* gemacht.

Hinzufügen von Elementen in ein Queue-Objekt.

```
Dim quMyQueue As New Queue()
```

Erstellt eine Referenz auf ein neues *Queue* -Objekt.

Danach können diesem *Queue* -Objekt mit der Methode *.Enqueue()* neue Elemente hinzugefügt werden

```
quMyQueue.Enqueue("Müller")
quMyQueue.Enqueue("Meier")
quMyQueue.Enqueue("Schulze")
quMyQueue.Enqueue("Mustermann")
quMyQueue.Enqueue("Musterfrau")
quMyQueue.Enqueue("Letzter")
```

Auslesen eines Queue-Objekt.

Sollen Elemente aus der Queue() wieder ausgelesen werden, so wird das mit der Methode *.Dequeue()* gemacht. Diese Methode gibt die Elemente als eine FiFo Sequenz zurück. Mit der Methode *.MoveNext()* wird zum einen auf das nächste Element in der Liste zugegriffen, zum anderen sorgt die Methode dafür, das nicht über das letzte Element hinaus zugegriffen wird.

```
' Queue auslesen, Elemente werden dabei gelöscht
While quMyQueue.GetEnumerator.MoveNext
    Console.WriteLine(quMyQueue.Dequeue.ToString)
End While
```

Gleichzeitig entfernt die Methode *.Dequeue()* auch die durchlaufene Elemente.