

# Datensammler

## Neue Datenstrukturen im .NET Framework – Teil 1

von Marcel Gnoth

Beim Einstieg in eine neue Programmierumgebung kommt jeder an den Punkt, an dem er Daten verwalten muss. Als .NET-Entwickler findet man recht schnell den *System.Collection*-Namespace, aber dann, oh weh! Die Qual der Wahl. Da gibt es Hashtables, ArrayLists, ListDictionaries, HybridDictionaries und, und, und. In diesem zweiteiligen Artikel möchte ich Ihnen einen Überblick über die verschiedenen Datenstrukturen des Frameworks geben und zeigen, wie Sie eigene Collection-Klassen mit C# und VB.NET implementieren können.

### Die gute alte Zeit...

Die VB6-Welt war recht überschaubar. Da gab es ein Collection-Objekt und wer wollte, konnte noch das Dictionary-Objekt der Scripting Runtime einsetzen. Das war es dann auch schon. Die C++-Fraktion konnte noch eine Reihe von Datenstrukturen, die die MFC-Klassenbibliothek zur Verfügung stellte, benutzen. Doch jetzt ist die .NET-Sonne über uns aufgegangen und für alle .NET-Sprachen gibt es nun einheitliche Datenstrukturen die das .NET Framework zur Verfügung stellt. Zu finden sind die Klassen in den Namespaces *System.Collection* und *System.Collection.Specialized*.

Was ist denn nun eine Collection-Klasse? Eine Collection-Klasse muss unter .NET eine Reihe von Interfaces implementieren, die Methoden zum Hinzufügen, Entfernen, Zugriff und Durchlaufen der Sammlung definieren. Alle Collection-Klassen des Frameworks implementieren mindestens eines der Interfaces. In Abbildung 1 finden Sie eine Übersicht über die Interfaces und die implementierenden Collection-Klassen, in Tabelle 1 finden Sie eine Zusammenstellung der Methoden und Eigenschaften der Interfaces. Diese Interfaces wollen wir uns im Folgenden anschauen.

### IEnumerable und IEnumerator

Durch *IEnumerable* wird eine Collection-Klasse eine aufzählbare Menge. Die einzige

Methode des Interfaces ist *GetEnumerator* und liefert ein *IEnumerator*-Objekt zurück. Dieses *IEnumerator*-Objekt ermöglicht der *foreach*-Schleife, über eine Datenmenge zu iterieren. *IEnumerator* verfügt über die Methode *MoveNext* und die Eigenschaft *Current*. Die Details der beiden Interfaces und deren Implementierung zeige ich Ihnen, wenn es um das Erstellen eigener Collection-Klassen geht.

### ICollection

Dieses Interface definiert eine abzählbare Menge, die *Count*-Property gibt die Anzahl der Elemente in der Collection zu-

rück. Dazu kommen zwei Properties für die Synchronisation in Applikationen mit mehreren Threads. Die *CopyTo*-Methode kopiert die Elemente in ein Array.

### ICollection und IDictionary

Die meisten Collection-Klassen können in zwei Gruppen eingeteilt werden. Die eine implementiert das *ICollection*- und die andere das *IDictionary*-Interface. Auf *ICollection*-Collections wird über einen Index zugegriffen, auf *IDictionary*-Collections über einen Schlüssel. Beide Interfaces definieren Methoden zum Einfügen, Löschen und Suchen von Elementen.

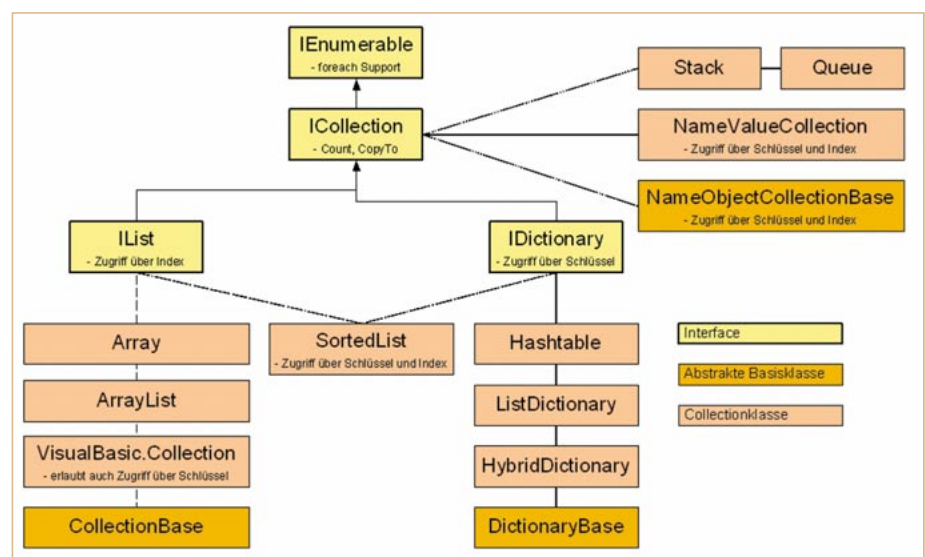


Abb. 1: Interfaces und Collection-Klassen des .NET Frameworks

Die *IDictionary*-Collection speichert Schlüssel-Wert-Paare. Auf die Menge aller Schlüssel kann über die *Keys*-Property und auf alle Werte über die *Values*-Property zugegriffen werden. Im Gegensatz zur alten VB-Collection enthält das Dictionary nicht direkt die Referenzen, sondern *DictionaryEntry*-Elemente. Ein *DictionaryEntry* hat eine *Key*- und eine *Value*-Property. Iterieren Sie mit *foreach* über ein Dictionary, dann iterieren Sie über eine Menge von *DictionaryEntry*-Elementen.

### Einige Bemerkungen am Anfang....

Zunächst einmal einige Merkmale. Alle Collections sind im Framework 0-basiert, die lästige Suche, ob 0- oder 1-basiert entfällt damit. Die meisten Collection-Klassen verfügen über einen Konstruktor und Methoden (*AddRange*), die als Argument *ICollection*-Objekte akzeptieren. So kann die Collection mit einem Aufruf gefüllt werden. Mit *RemoveRange* können mehrere Elemente auf einmal gelöscht werden. Collection-Klassen können auch als *Data-Source* eingesetzt werden.

Für den Zugriff auf ein bestimmtes Element einer Collection werden so genannte Indexer benutzt. Die Syntax in VB.NET und C# ist dabei unterschiedlich:

```
VB: myObj = Col.Item(7)
C#: myObj = Col[7]
```

### Überblick

In Tabelle 2 finden Sie eine Zusammenfassung der Collection-Klassen des Frameworks. Die Spalten *Index* und *Key* zeigen, wie Sie auf Elemente der Klasse zugreifen können. Nicht bei allen Collections muss der Schlüssel (*Key*) eindeutig sein, wie sie in Spalte vier sehen, und welche Datentypen Sie in der Collection speichern können, steht in der Spalte *Datentyp*. Einige Collections können ihre Elemente sortieren.

### Array

Ein Array ist immer noch die schnellste, aber auch unflexibelste Struktur. Mit *Rank* können Sie die Anzahl der Dimensionen, mit *Length* die absolute Anzahl der Elemente ermitteln. *GetLength* liefert die Anzahl der Elemente einer Dimension. Arrays können ihre Elemente sortieren, doch dazu später mehr. Da Arrays *IList* implementieren, können sie an viele Methoden des Frameworks als Argumente übergeben werden. Ihre Größe ist nicht veränderbar:

```
VB
Dim row() As Integer = {0, 1, 2, 3}
Dim row(4) As Integer

//C#
long[] row = {0, 1, 2, 3};
long[] row = new long[4];
long[] row = new long[] {0, 1, 2, 3};

foreach (int ID in colArr){
    ID=4;
}
```

### ArrayList

Die *ArrayList* ist ein eindimensionales Array, das wachsen und schrumpfen kann. Initial hat sie eine Größe von 16 Elemente, wie Sie über die *Capacity*-Property leicht feststellen können. Wird das 17. Element hinzugefügt, dann wächst die *ArrayList* um weitere 16 Elemente. *TrimToSize* gibt nicht mehr benötigten Speicherplatz frei. Die *ArrayList* implementiert *IList* und

IEnumerable	
GetEnumerator	Liefert ein Objekt vom Typ <i>IEnumerator</i> . Die <i>foreach</i> -Schleife holt sich als erstes so ein Objekt.
IEnumerator	
Current	Gibt der <i>foreach</i> -Schleife die aktuelle Objektreferenz.
MoveNext	Ändert internen Positionszeiger, sodass die <i>Current</i> -Property auf das nächste Objekt zeigt.
Reset	Setzt den internen Zähler so zurück, dass nach dem Aufruf der <i>MoveNext</i> -Methode, die <i>Current</i> -Property auf das erste Objekt zeigt.
ICollection	
CopyTo	Kopiert die Objektreferenzen in ein Array.
Count	Liefert die Anzahl der Elemente.
IsSynchronized	Synchronisation bei Anwendungen mit mehreren Programmthreads.
SyncRoot	
IList	
Add	Element hinzufügen.
Clear	Alle Elemente löschen.
Contains	Enthält die Collection ein bestimmtes Objekt?
IndexOf	Liefert den Index eines bestimmten Objekts.
Insert	Fügt Elemente an einer bestimmten Position ein.
IsFixedSize	Können Elemente hinzugefügt werden?
IsReadOnly	Können Referenzen geändert werden?
Remove	Löschen eines Objekte aus der Collection.
RemoveAt	Löschen eines Objekts an einer bestimmten Position.
this[], Item()	Zugriff auf ein Element an einer bestimmten Position.
IDictionary	
Add	Element hinzufügen.
Clear	Alle Elemente löschen.
Contains	Enthält die Collection ein bestimmtes Objekt?
GetEnumerator	Gibt <i>IEnumerator</i> -Objekt zurück.
IsFixedSize	Können Elemente hinzugefügt werden?
IsReadOnly	Können Referenzen geändert werden?
Keys	Liefert Collection aller Schlüssel der Collection.
Remove	Löschen eines Objekts aus der Collection.
this[], Item()	Zugriff auf ein Element über einen Schlüssel.
Values	Liefert Collection aller Objektreferenzen.

Tabelle 1: Die Interfaces der Collection-Klassen

deshalb erfolgt der Zugriff auf ihre Elemente über einen Index, und wie das Array kann sie ihre Elemente sortieren und eine binäre Suche auf ihnen ausführen. Sie ist eine der schnellsten Datenstrukturen, die Sie immer dann einsetzen sollten, wenn Sie keinen Zugriff über einen Schlüssel benötigen.

Und weil die ArrayList so schön ist, gibt es eine Möglichkeit, aus einem beliebigen *IList*-Objekt eine ArrayList zu generieren. Die statische Funktion *Adapter* erzeugt eine ArrayList aus einer anderen Collection. So können dann die Methoden der ArrayList auch auf andere Collections angewendet werden:

```
//lstDemos ist eine ListBox die bereits Elemente enthält
ArrayList arl = ArrayList.Adapter(lstDemos.Items);
//die Elemente der ListBox sind jetzt sortiert
arl.Sort();
```

## Hashtable

Wenn Sie viele Schlüssel-Wert-Paare verwalten müssen, dann ist die Hashtable der beste Kandidat. Für den übergebenen Schlüssel wird ein Hashcode berechnet

und anhand dieses Codes wird das Element in die Tabelle einsortiert. Schlüssel können nicht nur Strings, sondern auch Objekte sein. Durch diesen Hashcode ist es sehr schnell möglich, ein bestimmtes Element über seinen Schlüssel zu finden, da nicht die ganze Tabelle durchsucht werden muss, sondern die Position über den Hashcode ermittelt wird. Dadurch sind die Elemente aber nicht in der Reihenfolge ihres Einsortierens in der Hashtable angeordnet, sondern anhand ihres Hashcodes.

Bei *foreach* müssen Sie darauf achten, dass Sie über eine Menge von *Dictionary Entries* iterieren. Wollen Sie direkt über die Werte iterieren, dann nutzen Sie die *Values*-Property.

```
foreach (DictionaryEntry e in colDic) {
    Console.WriteLine("{0}/{1}", e.Key, e.Value);
}
foreach (Customer c in colCustDB.Values) {
    Console.WriteLine(c.Caption);
}
```

## ListDictionary und HybridDictionary

Das *ListDictionary* ist eine einfache verkettete Liste und der Verwaltungsauf-

wand ist verglichen mit der Hashtable geringer. Dafür wird das *ListDictionary* bei einer größeren Anzahl von Elementen sehr langsam. Empfohlen wird der Einsatz bei weniger als elf Elementen. Wenn Sie also sehr viele Collections mit sehr wenigen Elementen haben, dann würde sich das *ListDictionary* anbieten. Bereits ab 1.000 Elementen wird es spürbar langsamer. Nun, das wusste auch Microsoft und deshalb gibt es einen Hybriden. Das *HybridDictionary* kann automatisch zur Laufzeit zu einer Hashtable werden, wenn eine bestimmte Anzahl an Elementen erreicht wird. Dadurch wird einmalig Rechenzeit für das Neuanlegen der Hashtable und das Kopieren der wenigen Elemente verbraucht. Dieser Aufwand dürfte allerdings kaum Performance-relevant sein. Beide Klassen sind im *System.Collections.Specialized*-Namespace zu finden.

## SortedList

Die *SortedList* ist ein besonderer Vertreter, sie implementiert sowohl *IList* als auch *IDictionary* und erlaubt deshalb den Zugriff auf Elemente sowohl über den Index

Name	Index	Key	Key eindeutig	Datentyp Value	Sortiert	Bemerkung
Array	Ja	-	-	Object	Ja	-
ArrayList	Ja	-	-	Object	Ja	schnellste Collection
Queue	-	-	-	Object	-	FIFO
Stack	-	-	-	Object	-	LIFO
Hashtable	-	Ja	Ja	Object	-	schnellste Key-basierte Collection
SortedList	Ja	Ja	Ja	Object	Ja	Hybrid aus Hashtable und Array, sortiert nach den Keys
ListDictionary	-	Ja	Ja	Object	-	verkettete Liste, nur für kleine Anzahl an Elementen
HybridDictionary	-	Ja	Ja	Object	-	verwaltet kleine Mengen in ListDictionary, große in Hashtable
NameValueCollection	Ja	Ja	Nein	String	-	speichert unter einen String-Key (Name) mehrere String-Werte
StringCollection	Ja	-	-	String	-	Index-basiert
StringDictionary	-	Ja	Ja	String	-	Schlüssel-basiert
DictionaryBase	-	Ja	Ja	Typed	-	abstrakte Basisklasse, intern HashTable
CollectionBase	Ja	-	-	Typed	Ja	abstrakte Basisklasse, intern ArrayList
NameObject-CollectionBase	Ja	Ja	Nein	Typed	-	abstrakte Basisklasse, intern Hashtable, Keys sind String, bei Zugriff über Key erhalte ich bei Mehrfachvorkommen eines Keys nur das erste Objekt, sonst Index benutzen
BitArray	-	-	-	-	-	spezielle Bitoperationen, Referenz-Typ
BitVektor	-	-	-	-	-	32 Bits, Wert-Typ
DataSet	-	-	-	-	-	ADO.NET
XMLDocument	-	-	-	-	-	-
VisualBasic.Collection	Ja	Ja	Ja	Object	-	Referenz hinzufügen: Microsoft.VisualBasic, 1-basiert

Tabelle 2: Überblick über die Collection-Klassen des Frameworks

als auch über einen Schlüssel. Intern verwaltet sie zwei Arrays, eins für die Schlüssel und eins für die Werte. Die Elemente sind nach dem Schlüssel vom Typ *Object* sortiert. Wie sie auf die Sortierung Einfluss nehmen können, zeige ich Ihnen weiter unten. Das Einfügen von Elementen ist auf Grund der Sortierung um den Faktor 40 langsamer als bei einer Hashtable. Der Zugriff auf ein Element über den Key ist um den Faktor 10 langsamer als bei einer Hashtable. Der Zugriff über den Index ist vergleichbar mit der ArrayList. Spürbar werden diese Unterschiede allerdings erst ab 1.000 Elementen. Dafür bietet die SortedList eine ganze Reihe von Möglichkeiten, auf ihre Elemente zuzugreifen (*ContainsKey*, *ContainsValue*, *GetByIndex*, *GetKey*, *IndexOfValue*, *IndexOfKey*, ...).

### VisualBasic.Collection

Die gute alte VB-Collection, die einen Zugriff sowohl über einen Schlüssel als auch über den Index erlaubt, ist die einzige Collection, deren Index mit 1 beginnt. Sie verhält sich wie die alte VB6-Collection, ist langsamer als eine Hashtable und schnell

er als eine SortedList, hat aber dafür nur wenige Zugriffsmöglichkeiten auf ihre Elemente. Interessant wird sie, wenn Sie eine Collection brauchen, auf die Sie sowohl über Index als auch über Schlüssel zugreifen müssen und ihnen Performance wichtiger als die Flexibilität der SortedList ist. Wenn Sie die Collection in C# benutzen wollen, dann müssen Sie erst eine Referenz auf *Microsoft.VisualBasic* setzen und dann den Namespace importieren. Diese Collection ist vor allem aus Kompatibilitätsgründen im Framework zu finden. Sie implementiert *IList*, aber nicht *IDictionary*, obwohl sie einen Zugriff über einen Schlüssel erlaubt.

### String Dictionary / Collection

*StringDictionary* und *StringCollection* sind zwei speziell auf Strings zugeschnittene Datenstrukturen. *StringCollection* erlaubt einen Zugriff über Index und *StringDictionary* über einen Schlüssel. Sie entsprechen einer ArrayList bzw. Hashtable für Strings.

### Queue / Stack

Die *Queue* funktioniert nach dem *First In First Out* (FIFO) Prinzip und kann beliebige Objektreferenzen aufnehmen. Der *Stack* funktioniert genauso, nur anders herum: *Last In First Out* (LIFO). Mit *EnqueuePush* werden Elemente hinzugefügt, mit *DequeuePop* werden Elemente entnommen. Da beide *IEnumerable* implementieren, können sie mit *foreach* durchlaufen werden. *Peek* liest das nächste Element, ohne es aus der Struktur zu entnehmen, und mit *Contains* können Sie prüfen, ob sich ein bestimmtes Element in der Struktur befindet.

### BitArray / BitVector

Beide Klassen werden zur Verwaltung von Bitfeldern eingesetzt. Sie ermöglichen logische Operationen auf den Bitfeldern. Der Unterschied zwischen beiden ist, dass der *BitVector* ein Wert-Typ ist und auf dem Stack abgelegt wird. Er kann höchstens 32 Bits aufnehmen und ist effizienter beim Zugriff als das *BitArray*.

### NameValueCollection

Die *NameValueCollection* ist ein etwas ungewöhnlicher Vertreter. Sie können sowohl über Index als auch über Schlüssel

auf die Elemente zugreifen, aber die Schlüssel sind nicht eindeutig. Zwei Werte können unter dem gleichen Schlüssel abgelegt werden. Sowohl Schlüssel als auch Wert sind beide vom Typ *String*. Die Daten werden sozusagen gruppiert nach Schlüssel gespeichert. Greifen Sie über den Schlüssel auf die *NameValueCollection* zu und der Schlüssel wurde mehrfach vergeben, dann erhalten Sie alle Strings, die unter diesem Schlüssel abgelegt wurden durch Kommata getrennt:

```
NameValueCollection nv = new NameValueCollection();
nv.Add("Key4", "Julia"); nv.Add("Key5", "Karla");
nv.Add("Key1", "Anna");
nv.Add("Key1", "Barbara"); nv.Add("Key1", "Claudia");
nv.Add("Key2", "Doris"); nv.Add("Key2", "Erna");
```

Key4: Julia

Key5: Karla

Key1: Anna, Barbara, Claudia

Key2: Doris, Erna

### Selbst gemacht schmeckt es immer noch am besten

Die meisten oben genannten Collection-Klassen können Objektreferenzen eines beliebigen Typs speichern. In Objektmodellen möchten man aber streng typisierte Collection-Klassen haben, die nur eine Art von Objekten speichern. Zu VB6-Zeiten konnte man nur über Aggregation eine solche Collection-Klasse schreiben. Intern wurde eine VB-Collection verwaltet, die die Objektreferenzen aufgenommen hat. Die aktuelle Version von VB verfügt nun endlich auch über Vererbung und das Framework bietet vier abstrakte Basisklassen an, um eigene Collection-Klassen zu programmieren:

- *ReadOnlyCollectionBase*
- *CollectionBase*
- *DictionaryBase*
- *NameObjectCollectionBase*

Hinter der *CollectionBase* verbirgt sich eine *ArrayList* und hinter dem *DictionaryBase* eine *Hashtable*. Mit den Eigenschaften *InnerList* bzw. *InnerDictionary* können Sie auf diese Strukturen zugreifen. Sie leiten eine Ihrer Klassen von einer dieser abstrakten Klassen ab und brauchen nur noch einige Methoden zu überschreiben bzw. zu implementieren und fertig ist die streng typisierte Collection-Klasse.

#### Listing 1

```
//C# Beispiel
using System.Collections;
namespace DataStruct {
public class CTestColBase: CollectionBase {
public void Add(CCustomer Cu) {
this.InnerList.Add(Cu);
}
public CCustomer this[int x] {
get {
return (CCustomer) InnerList[x];
}
}
}
}

REM VB-Beispiel
Imports System.Collections
Public Class CTestColBase
Inherits CollectionBase
Public Sub Add(ByVal Cu As CCustomer)
Me.InnerList.Add(Cu)
End Sub
Public ReadOnly Property Item(ByVal x As Integer) As CCustomer
Get
Return CType(InnerList(x), CCustomer)
End Get
End Property
End Class
```

## Listing 2

```
using System.Collections;
namespace DataStruct {
public class CTestDicBase: DictionaryBase {
//Strong Typed Add
public void Add(CCustomer Cu, string k){
this.InnerHashtable.Add(k,Cu);
}
//Indexer
public CCustomer this[string Key] {
get {
return (CCustomer) InnerHashtable[Key];
}
}
//Remove
public void Remove(string Key){
InnerHashtable.Remove(Key);
}
}
//Support für foreach
public System.Collections.ICollection Values {
get {
return this.InnerHashtable.Values;
}
}
}
}
```

## Listing 3

```
foreach (string k in no.Keys)
System.Diagnostics.Debug.WriteLine
(k+" "+no[k].Caption);
System.Diagnostics.Debug.WriteLine
("-----");

for (int x=0; x<no.Count; x++)
System.Diagnostics.Debug.WriteLine
(x.ToString()+" "+no[x].Caption);
System.Diagnostics.Debug.WriteLine
("-----");

K1: Lisa Fox, BAHRAIN
K1: Lisa Fox, BAHRAIN
K1: Lisa Fox, BAHRAIN
K2: Famke Henstridge, AUSTRIA
K2: Famke Henstridge, AUSTRIA
K3: Kelly Jensen, SLOVENIA
K4: Joan Crow, TANSANIA
-----
0: Lisa Fox, BAHRAIN
1: Diana Anders, POLAND
2: Anja Kling, GREECE
3: Famke Henstridge, AUSTRIA
4: Claire Hatcher, ROMANIA
5: Kelly Jensen, SLOVENIA
6: Joan Crow, TANSANIA
```

## CollectionBase

In Listing 1 finden Sie ein Beispiel, das in C# und in VB7 eine Collection-Klasse für *CCustomer*-Objekte, meine Kundenklasse, definiert. Beide Klassen erben von *CollectionBase*. Jetzt müssen nur noch die Methoden implementiert werden, die nur einen speziellen Datentyp zulassen sollen. Die *Add*-Methode akzeptiert nur *CCustomer*-Objekte und ruft dann ihrerseits die *Add*-Methode der *InnerList* auf. Für den Zugriff auf ein bestimmtes Element einer Collection ist ein so genannter Indexer verantwortlich. In C# wird er mit einer Property namens *this* gefolgt von eckigen Klammern und in VB mit einer Property *Item* realisiert. Eine Methode *RemoveAt* ist bereits durch *CollectionBase* implementiert, da diese immer einen Index erwartet und keinen speziellen Datentyp. Wollen Sie eine *Remove*-Methode, die nur einen bestimmten Objekttyp akzeptiert, dann müssen Sie diese analog zur *Add*-Methode implementieren. Darüber hinaus können Sie implementieren, was immer Ihnen sinnvoll erscheint, z.B. weitere Methoden der *ArrayList* nach draußen „durchschleifen“.

## DictionaryBase

Eine Collection-Klasse, die Zugriff über einen Schlüssel auf ihre Elemente gestattet, wird im Prinzip genauso programmiert (Listing 2). Sie implementieren eine *Add*-Methode und einen Indexer, die nur einen speziellen Datentyp zulassen. *Remove* müssen Sie auch implementieren, da Sie bei einer Hashtable nicht über den Index löschen können (*RemoveAt* bei einer *ArrayList*), sondern nur über einen Schlüssel vom Typ *Object*. Also implementieren Sie eine *Remove*-Methode mit dem gewünschten *Key*-Typ. Diese Methode ruft dann einfach nur die *Remove*-Methode der *InnerHashtable* auf. Damit wären Sie jetzt eigentlich fertig, können aber nur die obere *foreach*-Schleife des Listings aus dem Abschnitt „Hashtable“ und eine Laufvariable vom Typ *DictionaryEntry* benutzen. Wenn Sie als Laufvariable den speziellen Datentyp der Collection-Klasse verwenden wollen, dann müssen Sie die *Values*-Property der *Hashtable* durchlaufen (untere Schleife im genannten Listing). Damit Sie das können, müssen Sie die *Values*-Property nach draußen „durchreichen“.

## NameObjectCollectionBase

Diese Datenstruktur ähnelt der *NameValueCollection*. Intern werden die Daten in einer *Hashtable* gespeichert. Sie können aber sowohl über Index als auch über einen Schlüssel auf die Elemente zugreifen. Schlüssel sind wie bei der *NameValueCollection* vom Typ *String* und nicht eindeutig, aber die Werte können beliebige Objektreferenzen sein. Das heißt, sie können mehrere Objekte nach einem *String* gruppieren. Sind unter einem Schlüssel mehrere Objekte abgelegt, dann erhalten Sie stets nur das erste, welches unter diesem Schlüssel abgelegt wurde. Über den Index können Sie auf jedes einzelne Element zugreifen (siehe Listing 3). Ein Implementierungsbeispiel finden Sie in Listing 4.

## Ausblick

Im zweiten Teil des Artikels werde ich Ihnen zeigen, wie Sie die vorgegebenen Pfade des .NET Frameworks verlassen und Ihre eigenen Collection-Klassen implementieren können. Darüber hinaus werde ich Ihnen mit dem ADO.NET DataSet und XmlDocument zwei alternative Möglichkeiten der Datenverwaltung vorstellen.

Marcel Gnoth ist Senior Consultant bei der Berliner NTeam GmbH ([www.nteam.de](http://www.nteam.de)). Seine Arbeitsschwerpunkte liegen im Bereich COM, .NET und verteilte Informationssysteme. Sie erreichen ihn unter [mgnoth@nteam.de](mailto:mgnoth@nteam.de) oder [www.gnoth.net](http://www.gnoth.net). ●

## Listing 4

```
public class CCustomersNameObjectColBase:
NameObjectCollectionBase{
public void Add(CCustomer Cu, string name){
this.BaseAdd(name,Cu);
}
public void Remove(string name){
this.BaseRemove(name);
}
public void Remove(int x){
this.BaseRemoveAt(x);
}
public CCustomer this[string Key] {
get { return (CCustomer) this.BaseGet(Key); }
}
public CCustomer this[int Index] {
get { return (CCustomer) this.BaseGet(Index); }
}
}
```