

Kapitel 8

LINQ to Objects

In diesem Kapitel:

Einführung in LINQ to Objects	154
Der Aufbau einer LINQ-Abfrage	155
Kombinieren und verzögertes Ausführen von LINQ-Abfragen	161
Verbinden mehrerer Auflistungen zu einer neuen	165
Gruppieren von Auflistungen	168
Aggregatfunktionen	172

Einführung in LINQ to Objects

Nach der Lektüre des vorherigen Kapitels wissen Sie ja im Grunde genommen schon, wie LINQ to Objects funktioniert, was gibt es also in Sachen Einführung zu LINQ to Objects noch zu sagen? – Was die generelle, interne Funktionsweise anbelangt, im Grunde gar nichts. Was die Auswirkungen von LINQ to Objects auf Ihre tägliche Routine anbelangt vielleicht eine ganze Menge!

Letzten Endes geht es nämlich beim Einsatz von LINQ to Objects nicht darum, möglichst eine Datenbank mit Business-Objekten quasi im Managed Heap Ihrer .NET-Anwendung nachzubilden – auch wenn die Beispiele dieses Kapitels das vielleicht vermuten lassen.

Nein? – Nein!

LINQ to Objects soll es Ihnen ermöglichen, die im letzten Kapitel vorgestellte, an die SQL-Abfragesprache angelehnte Syntax auch für ganz andere Zwecke, an die Sie im Moment vielleicht noch gar nicht denken, in ihre Programme einzubauen. Doch dazu müssen Sie wissen, mit welchen Komponenten einer Abfrage Sie welche Ergebnisse erzielen können – und dabei soll Ihnen dieses Kapitel behilflich sein. Aus diesem Grund wählen wir als Beispielträger auch einfachste Objektauflistungen (übrigens exakt die des letzten Kapitels), damit Sie sich bei den Beispielen auf LINQ konzentrieren können und nicht von Funktionalitäten anderer Auflistungsklassen des Systems abgelenkt werden. Dennoch können Sie LINQ nicht nur für ihre selbstgeschriebenen Business-Klassen und -Auflistungen einsetzen. Natürlich lassen sich auch Verzeichnis-, Grafik- oder Steuerelement-Auflistungen mit LINQ bearbeiten und organisieren.

HINWEIS

Dieses Kapitel kann übrigens dabei keinen Anspruch auf Vollständigkeit erheben – dazu ist das Thema LINQ einfach zu mächtig und umfangreich. Es wird Ihnen aber auf jeden Fall ausreichend Informationen, Beispiele und Anregungen zum Thema liefern, so dass Sie Ihr Wissen durch eigenes Ausprobieren und aufmerksames Lesen der Online-Hilfe sicherlich perfektionieren können. Das Visual Basic 2008 Entwicklerbuch, das voraussichtlich gegen Ende des 2. Quartals erscheinen wird, hat zum Thema LINQ noch mehr Informationen und Beispiele parat.

Übrigens: Der Einsatz von LINQ kann immer dann sinnvoll sein, wenn Sie mit gleich welchen Auflistungen arbeiten müssen. Das gilt vor allen Dingen unter Beachtung des folgenden Aspektes, dem ich – wegen seiner Zukunftsträchtigkeit- und -wichtigkeit –, einen eigenen Abschnitt widmen will; sicherlich ein wenig spekulativ, aber deswegen nicht minder wahrscheinlich – jedenfalls nach meinem bescheidenen Ermessen!

Verlieren Sie die Skalierbarkeit von LINQ nicht aus den Augen!

Denken Sie daran: LINQ als *Engine* zur Verarbeitung von internen .NET-Auflistungen ist skalierbar. Worauf ich hinaus will, möchte ich an einem einfachen Denkmodell demonstrieren:

Stellen Sie sich vor, Sie benötigen in Ihrer Anwendung einen Algorithmus zur Sortierung einer Auflistung. Dann können Sie, was gleichermaßen unnötig und unklug wäre, einen eigenen Algorithmus dazu schreiben. Unnötig, weil es bereits ausreichend Sortieralgorithmen im .NET Framework für alle möglichen Auflistungstypen gibt, und unklug, weil Sie selbst dafür zuständig wären, Ihren Algorithmus zu optimieren, beispielsweise weil Sie bemerken, dass Sie – was passieren wird – es nur noch mit Multi-Core-Prozessoren zu tun haben, und Ihre Anwendung, weil sie ständig (aber ständig nur auf einem Core des Prozessors) sortiert, nur 50%, 25% oder gar 12,5% der gesamt zur Verfügung stehenden Prozessorleistung ausschöpft. Ihr Algorithmus arbeitet nämlich nicht multithreaded.

Es gibt bei LINQ bereits experimentelle Ansätze und Konzepte,¹ die durchzuführenden Operationen mit mehreren Prozessoren (oder eben mehreren Prozessor-Cores) zu parallelisieren – jedenfalls soweit das möglich ist. Eine LINQ-Abfrage würde dann eine Kombination von Erweiterungsmethoden »auslösen«, die ihre eigentliche Arbeit in mehrere Threads und damit mehrere Cores aufteilen – das Resultat wäre eindrucksvoll: Gleicher Programmaufwand, doppelte, vierfache oder sogar achtfache Leistung!

LINQ anstelle normaler, selbstgestrickter For/Each-Auflistungsverarbeitungsschleifen lohnt sich aus planungstechnischer Sicht also auf jeden Fall – mal ganz abgesehen davon, dass Ihr Coding-Aufwand auch geringer wird und damit die Fehleranfälligkeit Ihres Programms abnimmt.

Der Aufbau einer LINQ-Abfrage

LINQ-Abfragen beginnen stets² mit der Syntax

```
From bereichsVariable In DatenAuflistung
```

und offen gesagt: Wenn ich auch die Syntax aus Programmiersicht nachvollziehen konnte, so verstand ich zunächst nicht, warum es unter rein sprachlichen Aspekten ausgerechnet From heißen musste – was wird denn da eigentlich von der Bereichsvariablen genommen, entwendet oder verwendet?

Also begann ich, ein wenig im Internet zu recherchieren, und ich stellte fest, dass nicht nur ich mir, sondern sich auch englische Muttersprachler diese Frage stellten, und so bemühte ich mich um ein wenig mehr Hintergrundinfos zur Entstehungsgeschichte der LINQ-Syntax bei jemandem, von dem ich glaubte, dass er es wissen müsse – und ich hatte Erfolg:

Lisa Feigenbaum vom Visual Basic Team über die Syntax von LINQ-Abfragen

Je mehr ich darüber nachdachte, wieso ausgerechnet From (und vielleicht nicht Through oder For) eine LINQ-Abfrage einleitet, desto mehr ließ mich auch die Frage nicht mehr in Ruhe, wieso eine LINQ-Abfrage nicht mit Select eingeleitet würde – so, wie wir Entwickler es schließlich schon seit Jahren von SQL-Server-Abfragen gewohnt waren.

Ich stellte diese Frage Lisa Feigenbaum vom Visual Basic Team, und sie antwortete mir dazu Folgendes:

»From« deutet auf Quelle oder Ursprungsort hin. Der From-Ausdruck als solcher ist die Stelle, an der man die Abfragequelle bestimmt (beispielsweise den Datenkontext, eine Auflistung im Speicher, XML, etc.). Aus diesem Grund fanden wir diesen Terminus als angemessen. Darüber hinaus handelt es sich bei »From« auch um ein SQL-Schlüsselwort. Wir haben uns bei LINQ um größtmöglichen SQL-Erhalt bemüht, sodass LINQ-Neueinsteiger, die bereits über SQL-Erfahrung verfügten, die Ähnlichkeit bereits »erfühlen« konnten. Ein interessanter Unterschied zwischen LINQ und SQL, den wir jedoch einbauen mussten, ist die Reihenfolge [der Elemente] innerhalb des Abfrageausdrucks. Bei SQL steht Select vor From. In LINQ ist es genau umgekehrt. ▶

¹ <http://msdn.microsoft.com/msdnmag/issues/07/10/PLINQ/default.aspx?loc=de> stellt einen interessanten Artikel zu PLINQ (Parallel LINQ) zur Verfügung (Stand: 24.01.2007).

² Ausnahmen bilden reine Aggregat-Abfragen, die auch mit der Klausel Aggregate beginnen, wie im entsprechenden Abschnitt dieses Kapitels beschrieben.

Einer der großen Vorteile von LINQ ist, dass man mit verschiedenen Datenquellen in einem gemeinsamen Modell arbeiten kann. Innerhalb einer Abfrage arbeitet man nur mit Objekten, und wir können IntelliSense für diese Objekte zur Verfügung stellen. Der Grund dafür, dass wir From als erstes benötigen, ist, dass wir als erstes wissen müssen, von woher die Quelle stammt, über die wir die Abfrage durchführen, bevor wir die IntelliSense-Information zur Verfügung stellen können.³

Sprachlich zu verstehen ist From also im Kontext des gesamten ersten Abfrageteilausdrucks – also quasi From (bereichsVariable in Auflistung) – und bezieht sich *nicht* nur auf die Bereichsvariable, die hinter From steht. Und nachdem dieser Ausdruck nun auch sprachlich geklärt ist, fassen wir zusammen:

From leitet eine LINQ-Abfrage ein, und er bedeutet ins »menschliche« übersetzt: »Hier, laufe mal bitte durch alle Elemente, die hinter dem Schlüsselwort In stehen, und verwende die Bereichsvariable hinter From, um mit deren Eigenschaften bestimmen zu können, wie die Daten der Auflistung selektiert, sortiert, gruppiert und dabei in eine neue Elementliste (mit – bei Bedarf – Instanzen anderer Elementklassen) überführt werden können.

WICHTIG

Wichtig für die Profi-T-SQL-ler unter Ihnen ist übrigens: LINQ-Abfragen beginnen niemals mit Select, ja, sie müssen noch nicht einmal ein Select aufweisen. DELETE, INSERT und UPDATE gibt es im Übrigen auch nicht – auch nicht wie im nächsten Kapitel zu lesen, in *LINQ to SQL*.

Mit dem Wissen des vorherigen Kapitels schauen wir uns jetzt mal die folgenden beiden Abfragen an und überlegen uns, wie sie sich im Ergebnis unterscheiden.

BEGLEITDATEIEN

Im Verzeichnis `\Samples\Chapter08 - Linq\LinqToObjects` finden Sie eine Konsolen-Beispielanwendung, die mit Business-Objekten weitere Möglichkeiten von *LINQ to Objects* erklären soll. Das Programm entspricht wieder dem aus dem letzten Kapitel: Es generiert per Zufallsgenerator zwei Auflistungen mit Business-Objekten (eine Kunden-, eine Artikelbestellaufistung) die logisch zu einander in Relation stehen. Das vorherige 7. Kapitel erklärt mehr zum Aufbau des Beispiels.

OK. Wir schauen uns zunächst das folgende Listing des Beispielprogramms an, und das schaut so aus:

```
Sub LINQAbfragenAufbau()

    Console.WriteLine("Ergebnisliste 1:")
    Console.WriteLine("-----")

    Dim ergebnisListe = From adrElement In adrListe _
                        Order By adrElement.Nachname, adrElement.Vorname Descending _
                        Select KontaktId = adrElement.ID, _
                            adrElement.Nachname, _
                            adrElement.Vorname, _
                            PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort _
                        Where KontaktId > 50 And KontaktId < 100 And _
                            Nachname Like "L*"
```

³ Quelle: Lisa Feigenbaum in einer E-Mail vom 01.02.2008. Lisa Feigenbaum ist VB IDE Program Manager im Visual Basic-Team bei Microsoft. Unter <http://blogs.msdn.com/vbteam/default.aspx>, der Team Blog-Seite, erreichen Sie Lisa Feigenbaum und das Visual Basic Team (Stand 03.02.2008).

```

For Each ergebnisItem In ergebnisListe
    With ergebnisItem
        Console.WriteLine(.KontaktId & ": " & _
                           .Nachname & ", " & .Vorname & _
                           " - " & .PlzOrtKombi)
    End With
Next

Console.WriteLine()
Console.WriteLine("Ergebnisliste 2:")
Console.WriteLine("-----")

Dim ergebnisListe2 = From adrElement In adrListe _
    Where adrElement.ID > 50 And adrElement.ID < 100 And _
        adrElement.Nachname Like "L*" _
    Select KontaktId = adrElement.ID, _
        adrElement.Nachname, _
        adrElement.Vorname, _
        PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort _
    Order By Nachname, Vorname

For Each ergebnisItem In ergebnisListe2
    With ergebnisItem
        Console.WriteLine(.KontaktId & ": " & _
                           .Nachname & ", " & .Vorname & _
                           " - " & .PlzOrtKombi)
    End With
Next
End Sub

```

Sie sehen, dass sich die Abfragen syntaktisch und damit auf den ersten Blick schon einmal gar nicht gleichen. Und dennoch, wenn Sie das Beispiel starten, so sehen Sie ein Ergebnis etwa wie im folgenden Bildschirmauszug...

```

Ergebnisliste 1:
-----
81: Langenbach, Barbara - 78657 Wiesbaden
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn

Ergebnisliste 2:
-----
81: Langenbach, Barbara - 78657 Wiesbaden
76: Langenbach, Klaus - 61745 Dortmund
54: Lehnert, Katrin - 82730 Wiesbaden
96: Löffelmann, Barbara - 63122 Rheda
68: Löffelmann, Gabriele - 54172 Bad Waldliesborn

```

... und obwohl diese Liste – laut Codelisting – mal definitiv auf zwei verschiedenen Ereignisaufflistungen beruht, die aus zwei verschiedenen Abfragen entstanden sind, ist das Ergebnis dennoch augenscheinlich dasselbe. Zufall? Keinesfalls – im Gegenteil, pure Absicht.

Denn dieses Beispiel demonstriert sehr schön, wie LINQ-Abfragen im Code funktionieren und wie sehr sie im Grunde genommen nichts mit den klassischen SQL-Datenbankabfragen zu tun haben. Gut, zugegeben, Ähnlichkeiten sind vorhanden, aber das war's auch schon.

Nun lassen Sie uns mal beide Abfragen auseinandernehmen und dabei schauen, wieso so unterschiedlich formuliertes dennoch zum gleichen Ergebnis führen kann. Los geht's:

- Die erste Abfrage startet, wie jede LINQ-Abfrage, mit der `From`-Klausel, und diese definiert `adrElement` als *Bereichsvariable* für alle kommenden Parameter bis – so vorhanden – zum ersten `Select`. Die Bereichsvariable ist also bis zum ersten `Select`-Befehl die Variable, mit der quasi intern durch die komplette Auflistung hindurchiteriert wird, und an der damit auch die Eigenschaften bzw. öffentlichen Felder der Auflistungselemente »hängen«.
- Es folgt hier im Beispiel (aber natürlich nicht notwendigerweise) die Sortieranweisung `Order By`, der die Felder bzw. Eigenschaften, nach denen die Elemente der Auflistung sortiert werden sollen, als Argumente über die Bereichsvariable übergeben werden: das sind in diesem Beispiel `adrElement.Nachname` und `adrElement.Vorname`. Die Bereichsvariable dient also dazu, auf die Eigenschaften zuzugreifen, über die durch die `Order By`-Klausel festgelegt werden kann, nach welchen Feldern sortiert wird.

TIPP Wenn nichts anderes gesagt wird, sortiert `Order By` *aufsteigend*. Durch die Angabe des Schlüsselwortes `Descending` können Sie den Sortierausdruck abändern, so dass die Elementauflistung *absteigend* sortiert wird. Wollte man – um beim Beispiel zu bleiben – die erste Liste nach Namen aufsteigend und nach Vornamen absteigend sortieren, hieße der LINQ-Abfrageausdruck folgendermaßen:

```
Dim ergebnisListe = From adrElement In adrListe
                    Order By adrElement.Nachname, adrElement.Vorname Descending
                    Select KontaktId = adrElement.ID,
                           adrElement.Nachname,
                           adrElement.Vorname,
                           PlzOrtKombi = adrElement.PLZ & " " & adrElement.Ort
                    Where KontaktId > 50 And KontaktId < 100 And
                           Nachname Like "L*"
```

- Jetzt folgt ein `Select`, und Achtung: Wir haben ja im letzten Kapitel erfahren, dass `Select` dazu dient, ein Element einer Auflistung durch ein anderes zu ersetzen, das in einer andere Auflistung »landet«. Und dieses zu ersetzende Element wird eine Instanz einer neuen anonymen Klasse mit exakt den Feldern sein, die Sie hinter dem `Select` bestimmen.

TIPP Spätestens hier sehen Sie, dass ein `Select` für eine Abfrage, die sich auf nur eine Tabelle bezieht, und die keine Gruppierungen durchführt, eigentlich ein reiner Zeitkiller ist, denn Sie können natürlich mit *den* Ausgangselementen weiterarbeiten, die *ohnehin* schon da sind und in der späteren Ergebnisliste *schlimmstenfalls* ein paar redundante Informationen aufweisen. Anders als bei SQL-Abfragen benötigen Sie das `Select`-Schlüsselwort bei LINQ *nicht*, um Abfragen für Filterungen, Selektionen oder Gruppierungen einzuleiten, sondern *nur*, um eine *neue Auflistung* mit Elementen zu erstellen, die aus einem neuen, anonymen Typ bestehen, der nur die Eigenschaften offenlegt, die Sie als Feldnamen angeben. Beherzigen Sie deswegen auch das, was der nächste Abschnitt über `Select` zu berichten weiß!

- Nachdem wir nun das `Select` hinter uns gelassen und der Abfrage-Engine damit mitgeteilt haben, dass wir nur noch mit `KontaktId`, `Nachname`, `Vorname` und einem `PlzOrtKombi`-Feld weitermachen wollen, können sich weitere LINQ-Abfrageelemente auch nur noch auf diese, durch `Select` neu eingeführte anonyme

Klasse beziehen – dementsprechend würde hinter der `Where`-Klausel ein `adrElement.Id` als Argument nicht mehr funktionieren; `Select` hat `adrElement` der Auflistung schließlich durch Instanzen einer anonymen Klasse ersetzt, und die haben nur noch die Eigenschaft `KontaktID`. `Where` dient jetzt dazu, die neue Auflistung mit den Elementen der anonymen Klasse zu filtern: nur Elemente mit `KontaktID > 50` und `KontaktID < 100` sind mit von der Partie, so wie die Elemente, deren Nachname mit »L« beginnt.

Damit ist die erste Abfrage durch. Und jetzt schauen wir uns die zweite an.

- Hier geht es nach der obligatorischen Festlegung von Auflistung und Bereichsvariable, die als Element-iterator dienen soll, zunächst los mit der `Where`-Klausel, die die Elemente auf jene IDs einschränkt, die größer als 50 und kleiner als 100 sind. Die Einschränkung wird dann noch weiterführt, nämlich auf Nachnamen, die mit dem Buchstaben »L« beginnen. Hier kann `Where` direkt auf die Eigenschaften zurückgreifen, die über die Bereichsvariable erreichbar sind, denn sie entsprechen einem Element der Ausgangsauflistung.
- Das geht solange, bis es auch hier wieder ein `Select` gibt, was dem ein Ende setzt. Auch hier wird wieder ein Ersetzungsvorgang durchgeführt – unnötigerweise und nur zu Demonstrationszwecken –, der die Elemente der vorhandenen Auflistung durch neue einer anonymen Klasse ersetzt, die wiederum die angegebenen Eigenschaften haben.
- Das anschließende `Where` schränkt diese Auflistung wieder ein, und zwar prinzipiell in gleicher Weise, wie im ersten Beispiel – lediglich der Zeitpunkt, wann die Elemente oder welche Elemente eingeschränkt werden, ist ein anderer.

Eines ist in diesem Zusammenhang sehr wichtig zu erwähnen:

WICHTIG LINQ-Abfragen werden immer verzögert ausgeführt, und niemals direkt. Das bedeutet: Nicht die eigentliche Abfrage ist ausschlaggebend für das »Anstoßen« der Verarbeitung, sondern erst das erste Zugreifen auf die Ergebnisliste löst das Ausführen der eigentlichen Abfrage aus. Mehr dazu erfahren Sie im Abschnitt »Kombinieren und verzögertes Ausführen von LINQ-Abfragen«.

Die Ausführung von `Select` ergibt bei der Anwendung von nur einer Auflistung nur eingeschränkt Sinn. Das folgende Beispiel zeigt, wieso `Select` Rechenleistung kosten kann ohne einen nachvollziehbaren Nutzen dabei zu bringen.

Dieses Beispiel arbeitet mit einem Hochgeschwindigkeitszeitmesser, um die Laufdauer bestimmter Auswertungen zu messen. Dabei wird eine Auflistung mit 2.000.000 Kundenelementen einmal mit und einmal ohne `Select`-Abfrage ausgeführt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojektes folgendermaßen abzuändern:

Module LinqDemo

```
Private adrListe As List(Of Kontakt) = Kontakt.Zufallskontakte(2000000)
Private artListe As List(Of Artikel) '= Artikel.Zufallsartikel(adrListe)
```

Achten Sie darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

Die eigentliche Methode, die das Beispiel enthält, sieht wie folgt aus:

```

Sub SelectSpeedCompare()

    Dim hsp As New HighSpeedTimeGauge
    Console.WriteLine("Starte Test")
    hsp.Start()
    Dim ergebnisListe = From adrElement In adrListe _
                        Order By adrElement.Nachname, adrElement.Vorname _
                        Select KontaktId = adrElement.ID, _
                               adrElement.Nachname, _
                               adrElement.Vorname, _
                               PLZOrtKombi = adrElement.PLZ & " " & adrElement.Ort

    Dim ersteAnzahl = ergebnisListe.Count
    hsp.Stop()
    Dim dauer1 = hsp.DurationInMilliseconds

    hsp.Reset()
    hsp.Start()
    Dim ergebnisListe2 = From adrElement In adrListe _
                        Order By adrElement.Nachname, adrElement.Vorname

    Dim zweiteAnzahl = ergebnisListe2.Count
    hsp.Stop()
    Dim dauer2 = hsp.DurationInMilliseconds

    Console.WriteLine("Abfragedauer mit Select: " & dauer1 & " für " & ersteAnzahl & " Elemente.")
    Console.WriteLine("Abfragedauer ohne Select: " & dauer2 & " für " & zweiteAnzahl & " Elemente.")
End Sub

```

Das Ergebnis dieses Tests offenbart den Unterschied. Bei 2.000.000 Elementen kommt es ...

```

Starte Test
Abfragedauer mit Select: 16939 für 2000000 Elemente.
Abfragedauer ohne Select: 16061 für 2000000 Elemente.

```

... bis zum Unterschied von einer Sekunde. Natürlich ist das kein Wert, an dem man sich für die Praxis in irgendeiner Form orientieren sollte; das Beispiel ist – schon klar – zudem natürlich auch praxisfern, denn wer zwei Millionen Datensätze im Speicher hält, hat entweder eine Datenbankengine programmiert oder das Konzept von Datenbanken und Datenbankabfragen nicht verstanden. Aber es geht nur um Tendenzen und den Tipp, Prozessorleistung nicht an Stellen unnötig zu verheizen, an denen es nicht nötig ist.

Übrigens, und besser könnte diese Überleitung zum nächsten Abschnitt nicht sein, wenn Sie sich das vorherige Listing anschauen, sollte Ihnen etwas auffallen, das Sie erstaunen sollte. Sie sehen es nicht? OK – dann überlegen Sie sich mal, ob es Zufall ist, dass die Count-Eigenschaft noch innerhalb der Zeitmessung abgefragt wird. Denn: Das ist nicht nur bewusst so gemacht, anderenfalls würden Sie komplett andere Ergebnisse bekommen, denn wenn Sie das Listing folgendermaßen abändern ...

```

Sub SelectSpeedCompare()
.
.
.
    hsp.Stop()
    Dim dauer1 = hsp.DurationInMilliseconds
    Dim ersteAnzahl = ergebnisListe.Count

```

```

hsp.Reset()
hsp.Start()
Dim ergebnisListe2 = From adrElement In adrListe
                    Order By adrElement.Nachname, _adrElement.Vorname

hsp.Stop()
Dim dauer2 = hsp.DurationInMilliseconds
Dim zweiteAnzahl = ergebnisListe2.Count
.
.
.
End Sub

```

... erhalten Sie auf einmal das folgende Ergebnis!

```

Starte Test
Abfragedauer mit Select: 1 für 2000000 Elemente.
Abfragedauer ohne Select: 0 für 2000000 Elemente.

```

Ein Fehler? Nein – das ist genau das Ergebnis, was bei der Ausführung herauskommen muss!

Kombinieren und verzögertes Ausführen von LINQ-Abfragen

Eines vorweg: LINQ-Abfragen werden immer verzögert ausgeführt, die Überschrift könnte also insofern in die Irre führen, als dass sie impliziert, der Entwickler, der sich LINQ-Abfragen bedient, hätte eine Wahl. Er hat sie nämlich nicht.

Wann immer Sie eine Abfrage definieren, und sei sie wie die folgende ...

```

Dim adrListeGruppirt = From adrElement In adrListe _
                    Join artElement In artListe _
                    On adrElement.ID Equals artElement.IDGekauftVon _
                    Select adrElement.ID, adrElement.Nachname, _
                        adrElement.Vorname, adrElement.PLZ, _
                        artElement.ArtikelNummer, artElement.ArtikelName, _
                        artElement.Anzahl, artElement.Einzelpreis, _
                        Postenpreis = artElement.Anzahl * artElement.Einzelpreis _
                    Order By Nachname, ArtikelNummer
                    Where (PLZ > "0" And PLZ < "50000") _
                    Group ArtikelNummer, ArtikelName, _
                        Anzahl, Einzelpreis, Postenpreis _
                    By ID, Nachname, Vorname _
                    Into ArtikeListe = Group, Anzahl|Artikel = Count(), _
                        Gesamtpreis = Sum(Postenpreis)

```

... noch so lang; die Abfragen selbst werden nicht zum Zeitpunkt ihres »Darüberfahrens« ausgeführt. Im Gegenteil: In `adrListeGruppirt` wird – um bei diesem Beispiel zu bleiben – im Prinzip nur eine Art *Ausführungsplan* generiert, also eine Liste der Methoden, die nacheinander ausgeführt werden, sobald ein Element aus der (noch zu generierenden!) Ergebnisliste abgerufen wird, oder Eigenschaften bzw. Funktionen der Ergebnisliste abgerufen werden, die in unmittelbarem Zusammenhang mit der Ergebnisliste selbst stehen – wie beispielsweise die `Count`-Eigenschaft.

Und es wird noch besser: Verschiedene LINQ-Abfragen lassen sich so nacheinander »aufreihen«, wie das folgende Beispiel eindrucksvoll zeigt.

HINWEIS Achten Sie beim Einrichten dieses Beispiels darauf, die ersten Zeilen des Beispielprojektes folgendermaßen abzuändern:

Module LinqDemo

```
Private adrListe As List(Of Kontakt) = Kontakt.Zufallskontakte(500000)
Private artListe As List(Of Artikel) '= Artikel.Zufallsartikel(adrListe)
```

Achten Sie darauf, diese Zeilen für die anderen Beispiele wieder zurückzubauen.

```
Sub SerialLinqsCompare()

    Dim hsp As New HighSpeedTimeGauge
    Console.WriteLine("Starte Test")
    hsp.Start()
    Dim ergebnisListe = From adrElement In adrListe _
                        Order By adrElement.Nachname, adrElement.Vorname

    Dim ergebnisListe2 = From adrElement In ergebnisListe _
                        Where adrElement.Nachname Like "L*"

    ergebnisListe2 = From adrElement In ergebnisListe2 _
                    Where adrElement.ID > 50 And adrElement.ID < 200

    Dim ersteAnzahl = ergebnisListe2.Count
    hsp.Stop()
    Dim dauer1 = hsp.DurationInMilliseconds

    hsp.Reset()
    hsp.Start()
    Dim ergebnisListe3 = From adrElement In adrListe _
                        Order By adrElement.Nachname, adrElement.Vorname _
                        Where adrElement.Nachname Like "L*" And _
                        adrElement.ID > 50 And adrElement.ID < 200

    Dim zweiteAnzahl = ergebnisListe3.Count
    hsp.Stop()
    Dim dauer2 = hsp.DurationInMilliseconds

    Console.WriteLine("Abfragedauer serielle Abfrage: " & dauer1 & " für " & ersteAnzahl & "
        Ergebniselemente.")
    Console.WriteLine("Abfragedauer kombinierte Abfrage: " & dauer2 & " für " & zweiteAnzahl & "
        Ergebniselemente.")

End Sub
```

Der zweite, in Fettschrift markierte Block entspricht im Grunde genommen dem ersten – nur das hier Abfragen hintereinandergeschaltet, aber eben nicht ausgeführt werden. Die eigentliche Ausführung findet statt, wenn auf die zu entstehende Elementauflistung zugegriffen wird – im Beispiel also die Count-Eigenschaft der »Auflistung« abgerufen wird, die die Anzahl der Elemente nur dann zurückgeben kann, wenn es eine Anzahl an Elementen gibt.

Dass sich die beiden Ausführungspläne nicht nennenswert unterscheiden, zeigt das folgende Ergebnis ...

```
Starte Test
Abfragedauer serielle Abfrage: 3531 für 17 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 3561 für 17 Ergebniselemente.
```

... das mit 30ms Unterschied bei 500.000 Elementen wirklich nicht nennenswert ist.

WICHTIG Die Ausführungspläne, die Sie durch die Abfragen erstellen, werden jedes Mal ausgeführt, wenn Sie auf eine der Ergebnislisten zugreifen. Wiederholen Sie die letzte fettgeschriebene Zeile im obigen Listing ...

```
.
.
    Dim zweiteAnzahl = ergebnisListe3.Count
    zweiteAnzahl = ergebnisListe3.Count
    hsp.Stop()
    Dim dauer2 = hsp.DurationInMilliseconds
.
.
```

... ergibt sich das folgende Ergebnis:

```
Starte Test
Abfragedauer serielle Abfrage: 3675 für 12 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7104 für 12 Ergebniselemente.
```

Faustregeln für das Erstellen von LINQ-Abfragen

Die Faustregeln für das Erstellen von Abfragen lauten:

- LINQ-Abfragen bestehen nur aus Ausführungsplänen – die eigentlichen Abfragen werden durch ihre Definition nicht ausgelöst!
- Wenn das Ergebnis einer Abfrage als Ergebnisliste Gegenstand einer weiteren Abfrage wird, kommt der erste Abfrageplan auch nicht zur Auslösung; beide Abfragepläne werden miteinander kombiniert.
- Erst der Zugriff auf ein Element der Ergebnisauflistung (über For/Each oder den Indexer) löst die Abfrage und damit das Erstellen der Ergebnisliste aus.
- Ein erneuter Zugriff auf eine elementeabhängige Eigenschaft oder auf die Elemente selbst löst – und das ist ganz wichtig – abermals das Erstellen der Ergebnisliste aus. Das gilt auch für Abfragen, die durch mehrere Abfragen kaskadiert wurden.

Kaskadierte Abfragen

Das Beispiel, was ich Ihnen im letzten Listing vorgestellt habe, hat nämlich noch zwei weitere, auskommentierte Zeilen, die die Kaskadierungsfähigkeit (das Hintereinanderschalten) von Abfragen eindrucksvoll demonstrieren. Wenn Sie das Ergebnis dieser beiden zusätzlichen Zeilen des Listings ...

```
adrListe.Add(New Kontakt(51, "Löffelmann", "Klaus", "Wiedenbrücker Straße 47", "59555", "Lippstadt"))
Console.WriteLine("Elemente der seriellen Abfrage: " & ergebnisListe2.Count)
```

... verstanden haben, dann haben Sie auch das Prinzip von LINQ verstanden!

```
Starte Test
Abfragedauer serielle Abfrage: 3515 für 7 Ergebniselemente.
Abfragedauer kombinierte Abfrage: 7101 für 7 Ergebniselemente.
Elemente der seriellen Abfrage: 8
```

Hier wird der Originalauflistung ein weiteres Element hinzugefügt, das exakt der Kriterienliste der kaskadierten Abfrage entspricht. Durch das Abfragen der Count-Eigenschaft von `ergebnisListe2` wird die *komplette* Abfragekaskade ausgelöst, denn von vorher 7 Elementen befinden sich anschließend 8 Elemente in der Ergebnismenge, was nicht der Fall wäre, würde LINQ nur die letzte Ergebnismenge, nämlich `ergebnisListe2` selbst auswerten, denn dieser haben wir das Element nicht hinzugefügt.

Gezieltes Auslösen von Abfragen mit ToArray oder ToList

Nun haben wir gerade erfahren, dass Abfragen, bei jedem Zugriff auf die Ergebnisliste mit `For/Each` oder direkt über eine Elementeigenschaft bzw. den Index zu einer neuen Ausführung des Abfrageplans führen. Das kann, wie im Beispiel des vorherigen Abschnittes, durchaus wünschenswert sein – in vielen Fällen können sich daraus aber echte Performance-Probleme entwickeln.

Aus diesem Grund implementierten die Entwickler von LINQ spezielle Methoden, die eine auf `IEnumerable` basierende Ergebnisliste wieder in eine »echte« Auflistung bzw. in ein »echtes« Array umwandeln können.

Am häufigsten wird dabei sicherlich die Methode `ToList` zur Anwendung kommen, die das Ergebnis einer wie auch immer gearteten LINQ-Abfrage in eine generische `List(Of t)` umwandelt – und dabei ist es völlig gleich, ob es sich um eine *LINQ to Objects*, *LINQ to SQL*, *LINQ to XML* oder eine *LINQ to sonst zu was*-Abfrage handelt. Das Ausführen von `ToList` auf eine solche Ergebnisliste hat zwei Dinge zur Folge:

- Die LINQ-Abfrage wird ausgeführt.
- Eine `List(Of {Ausgangstyp|Select-Anonymer-Typ})` wird zurückgeliefert.

Die Elemente, die anschließend zurückkommen, sind völlig ungebunden – Sie können sie anschließend sofort indizieren, mit `For/Each` durchiterieren, ihre `Count`-Eigenschaft abfragen wie Sie wollen – mit der ursprünglichen LINQ-Abfrage haben sie nichts mehr zu tun.

`ToList` funktioniert dabei denkbar einfach:

```
Dim reineGenerischeListe = ergebnisListe.ToList
```

Und `ToList` ist auch nicht die einzige Methode, mit der Sie eine Ergebnisliste von ihrer Ursprungsabfrage trennen können. Das können Sie – in Form von Ergebnislisten unterschiedlichen Typs – auch mit den folgenden Methoden machen. `t` ist dabei immer der Typ eines Elements der Ausgangsauflistung oder ein anonymer Typ, der in der Abfrage beispielsweise durch die `Select`-Klausel entstanden ist.

- `ToList`: Überführt die Abfrageergebnisliste in eine generische `List(Of t)`.
- `ToArray`: Überführt die Abfrageergebnisliste in ein Array vom Typ `t`.

- `ToDictionary`: Überführt die Abfrageliste in eine generische Wörterbuchauflistung vom Typ `Dictionary(Of t.key, t)`. `t.key` muss dabei durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa

```
Dim reineGenerischeListe = ergebnisListe.ToDictionary(Function(einKontakt) einKontakt.ID)
```

um beim Beispiel zu bleiben, und die ID zum Nachschlüssel zu machen.

- `ToLookup`: Überführt die Abfrageliste in eine generische Lookup-Auflistung vom Typ `Lookup(Of t.key, t)`. `t.key` muss dabei durch die Angabe eines Lambda-Ausdrucks festgelegt werden, also etwa

```
Dim reineGenerischeListe = ergebnisListe.ToLookup(Function(einKontakt) einKontakt.ID)
```

HINWEIS

Die `ToLookup`-Methode gibt ein generisches Lookup-Element zurück, ein `1:n`-Wörterbuch, das Auflistungen von Werten Schlüssel zuordnet. Ein Lookup unterscheidet sich von Dictionary, das eine `1:1`-Zuordnung von Schlüsseln zu einzelnen Werten ausführt.

Verbinden mehrerer Auflistungen zu einer neuen

LINQ kennt mehrere Möglichkeiten, Auflistungen miteinander zu verbinden. Natürlich ist es nicht unbedingt sinnvoll, das willkürlich zu tun: Die Auflistungen müssen schon in irgendeiner Form miteinander in logischer Relation stehen – dann aber kann man sich durch geschicktes Formulieren eine Menge an Zeit sparen. Wie eine solche Relation ausschauen kann, in der zwei Auflistungen zueinander stehen, demonstriert Ihnen der Abschnitt »Wie geht LINQ prinzipiell« des vorherigen Kapitels – und diese Relation soll auch noch mal Gegenstand der Beispiele dieses Abschnittes sein:

ID	Nachname	Vorname	Straße	PLZ	Ort
1	Heckhuis	Jürgen	Wiedenbrücker Str. 47	59555	Lippstadt
2	Wördehoff	Angela	Erwitter Str. 33	01234	Dresden
3	Dröge	Ruprecht	Douglas-Adams-Str. 42	55544	Ratingen
4	Dröge	Ada	Douglas-Adams-Str. 42	55544	Ratingen
5	Halek	Gaby „Doc“	Krankenhausplatz 1	59494	Soest

	IdGekauftVon	ArtikelNummer	ArtikelName	Kategorie	Einzelpreis	Anzahl
1	1	9-445	Die Tore der Welt	Bücher, Belletristik	19,90	2
3	3	3-537	Visual Basic 2005 - Das Entwicklerbuch	Bücher, EDV	59,00	2
3	3	3-123	SQL Server 2000 - So gelingt der Einstieg	Bücher, EDV	19,90	1
5	5	5-312	SQL Server 2008 - Das Profi-Buch	Bücher, EDV	39,90	2

Abbildung 8.1 Das LINQ-Beispielprogramm legt zwei Tabellen an, die nur logisch über die Spalten-ID miteinander verknüpft sind

Implizite Verknüpfung von Auflistungen

Die einfachste Möglichkeit, zwei Auflistungen zu gruppieren, zeigt die folgende Abbildung.

```

Sub Auflistungsvereinigung()
    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe

    For Each element In ergebnisliste
        element.
        
        with element
            Console.WriteLine(,adrElement.ID & ": " & .adrElement.Nachname & _
                ", " & .adrElement.Vorname & ": " & _
                .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
        End With
    Next
End Sub

```

Abbildung 8.2 Schon mit der From-Klausel können Sie zwei Auflistungen per LINQ miteinander kombinieren

From kombiniert zwei Auflistungen miteinander. Im Ergebnis erhalten Sie eine neue Auflistung, bei der jedes Element der ersten mit jedem Element der zweiten Auflistung kombiniert wurde. Die Ausgabe dieser Auflistung, die über zwei Eigenschaften verfügt, die jeweils Zugriff auf die Originalelemente gestatten, zaubert dann folgendes Ergebnis auf den Bildschirm (aus Platzgründen gekürzt):

```

6: Clarke, Christian: 1: Visual Basic 2008 - Das Entwicklerbuch
6: Clarke, Christian: 1: Das Herz der Hölle
6: Clarke, Christian: 2: Visual Basic 2005 - Das Entwicklerbuch
6: Clarke, Christian: 2: Das Vermächtnis der Tempelritter
6: Clarke, Christian: 3: Das Leben des Brian
6: Clarke, Christian: 3: Die Tore der Welt
.
.
.
7: Ademmer, Lothar: 1: Visual Basic 2008 - Das Entwicklerbuch
7: Ademmer, Lothar: 1: Das Herz der Hölle
7: Ademmer, Lothar: 2: Visual Basic 2005 - Das Entwicklerbuch
7: Ademmer, Lothar: 2: Das Vermächtnis der Tempelritter
7: Ademmer, Lothar: 3: Das Leben des Brian
7: Ademmer, Lothar: 3: Die Tore der Welt

```

An der Ergebnisliste können Sie erkennen, wie redundant und nicht informativ diese Liste ist, denn jeder Artikel der Artikelliste wird einfach und stumpf mit jedem Kontakt kombiniert.

Wichtiger wäre es, Zuordnungen ausdrücklich bestimmen zu können, um zu sagen, dass ein Artikel mit einer bestimmten ID auch nur dem logisch dazugehörigen Kontakt zugeordnet werden soll. Und das funktioniert folgendermaßen:

```

Sub Auflistungsvereinigung()
    Dim ergebnisliste = From adrElement In adrListe, artElement In artListe _
        Where adrElement.ID = artElement.IDGekauftVon

```

```

For Each element In ergebnisliste
  With element
    Console.WriteLine(.adrElement.ID & ": " & .adrElement.Nachname & _
                      ", " & .adrElement.Vorname & ": " & _
                      .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
  End With
Next
End Sub

```

In dieser Version werden durch die `Where`-Klausel nur die Elemente in die Auflistung übernommen, die die gleiche ID haben wie das korrelierende Element der anderen Auflistung (`IDGekauftVon`). Das Ergebnis, was dann zu sehen ist, macht natürlich viel mehr Sinn, da es eine Aussagekraft hat (nämlich: welcher Kunde hat welche Artikel gekauft):

```

7: Sonntag, Uwe: 7: Visual Basic 2008 - Das Entwicklerbuch
7: Sonntag, Uwe: 7: The Da Vinci Code
7: Sonntag, Uwe: 7: O.C., California - Die komplette zweite Staffel (7 DVDs)
7: Sonntag, Uwe: 7: Desperate Housewives - Staffel 2, Erster Teil
7: Sonntag, Uwe: 7: Die Rache der Zwerge
8: Vüllers, Momo: 8: Programmieren mit dem .NET Compact Framework
9: Tinoco, Daja: 9: Das Herz der Hölle
9: Tinoco, Daja: 9: Mitten ins Herz
9: Tinoco, Daja: 9: The Da Vinci Code
9: Tinoco, Daja: 9: Der Schwarm
9: Tinoco, Daja: 9: Desperate Housewives - Staffel 2, Erster Teil
9: Tinoco, Daja: 9: Harry Potter und die Heiligtümer des Todes
9: Tinoco, Daja: 9: Der Teufel trägt Prada
9: Tinoco, Daja: 9: O.C., California - Die komplette zweite Staffel (7 DVDs)
10: Lehnert, Michaela: 10: Abgefahren - Mit Vollgas in die Liebe
10: Lehnert, Michaela: 10: Das Herz der Hölle

```

HINWEIS

Die Verknüpfung zweier oder mehrerer Auflistungen mit `In` als Bestandteil der `From`-Klausel einer Abfrage nennt man implizite Verknüpfung von Auflistungen, da dem Compiler nicht ausdrücklich mitgeteilt wird, welche Auflistung auf Grund welchen Elementes mit einer anderen verknüpft wird. Eine ausdrückliche oder explizite Verknüpfung stellen Sie mit `Join` her, das im nächsten Abschnitt beschrieben wird. Explizit oder implizit hier im Beispiel ist aber letzten Endes einerlei – das Ergebnis ist in beiden Fällen dasselbe.

Explizite Auflistungsverknüpfung mit Join

Im Gegensatz zu impliziten Auflistungsverknüpfungen, die Sie, wie im letzten Abschnitt beschrieben, mit `In` als Teil der `From`-Klausel bestimmen, erlaubt Ihnen die `Join`-Klausel sogenannte explizite Auflistungsverknüpfungen festzulegen. Die generelle Syntax der `Join`-Klausel lautet:

```

Dim ergebnisliste = From bereichsVariable In ersterAuflistung _
                    Join verknuepfungselement In zweiterAuflistung _
                    On bereichsVariable.JoinKey Equals verknuepfungselement.ZweiterJoinKey

```

Join verknüpft die erste mit der zweiten Tabelle über einen bestimmten Schlüssel (JoinKey, ZweiterJoinKey), der beide Tabellen zueinander in Relation stellt. In unserem Beispiel wird für jede Bestellung in der Artikel-tabelle ein Schlüssel (ein *Key*, eine *ID*) definiert, der der Nummer der *ID* in der Kontakt-tabelle entspricht.

Im Vergleich zur impliziten Verknüpfung von Tabellen ändert sich im Ergebnis nichts; die Umsetzung des vorherigen Beispiels mit Join gestaltet sich folgendermaßen:

```
Sub JoinDemo()
    Dim ergebnisliste = From adrElement In adrListe _
                        Join artElement In artListe _
                        On adrElement.ID Equals artElement.IDGekauftVon

    For Each element In ergebnisliste
        With element
            Console.WriteLine(.adrElement.ID & ": " & .adrElement.Nachname & _
                              ", " & .adrElement.Vorname & ": " & _
                              .artElement.IDGekauftVon & ": " & .artElement.ArtikelName)
        End With
    Next
End Sub
```

Es gibt die Möglichkeit, mit der Klausel `GroupJoin` Verknüpfungen mehrerer Tabellen auf bestimmte Weise in Gruppen zusammenzufassen. Wie das funktioniert erfahren Sie im Abschnitt »Group Join« ab Seite 171.

Gruppieren von Auflistungen

Die Klausel `GroupBy` erlaubt es, Dubletten von Elementen einer oder mehrerer Auflistungen in Gruppen zusammenzufassen. Sie möchten also beispielsweise eine Auflistung von Kontakten nach Nachnamen gruppieren, und dann in einer geschachtelten Schleife durch die Namen und innen durch alle den Namen zugeordneten Kontakte iterieren. Mit der `GroupBy`-Klausel können Sie genau das erreichen, wie das folgende Beispiel zeigt:

```
Sub GroupByDemo()
    Dim ergebnisliste = From adrElement In adrListe _
                        Group By adrElement.Nachname Into Kontaktliste = Group _
                        Order By Nachname

    For Each element In ergebnisliste
        With element
            Console.WriteLine(element.Nachname)
            For Each kontakt In element.Kontaktliste
                With kontakt
                    Console.WriteLine(.ID & ": " & .Nachname & ", " & .Vorname)
                End With
            Next
            Console.WriteLine()
        End With
    Next
End Sub
```

Wörtlich ausformuliert hieße die Group By-Klausel dieses Beispiels: »Erstelle eine Liste aller Nachnamen (Group By adrElement.Nachname) und mache diese unter der Nachname-Eigenschaft in der Liste zugänglich.⁴ Fasse alle Elemente der Ausgangsliste in jeweiligen Auflistungen zusammen, die dem Nachnamen zugehörig sind (Into ... = Group), und mache diese Auflistung über die Eigenschaft Kontaktliste verfügbar.«

HINWEIS

Falls Sie keine Aliasbenennung der Gruppe vornehmen (der Ausdruck würde dann einfach

```
Dim ergebnisliste = From adrElement In adrListe _  
                    Group By adrElement.Nachname Into Group _  
                    Order By Nachname
```

heißen, würde die Eigenschaft, mit der Sie die zugeordneten Elemente erreichen können, einfach Group heißen.

Wenn Sie das Beispiel starten, sehen Sie das gewünschte Ergebnis, etwa wie im folgenden Bildschirmauszug (aus Platzgründen gekürzt):

```
Weichelt  
19: Weichelt, Anne  
39: Weichelt, Gabriele  
47: Weichelt, Uta  
69: Weichelt, Franz  
97: Weichelt, Hans  
  
Weigel  
37: Weigel, Lothar  
40: Weigel, Rainer  
43: Weigel, Lothar  
58: Weigel, Hans  
60: Weigel, Anja  
  
Westermann  
11: Westermann, Margarete  
21: Westermann, Alfred  
28: Westermann, Alfred  
41: Westermann, Alfred  
77: Westermann, Guido  
98: Westermann, José  
100: Westermann, Michaela  
  
Wördehoff  
14: Wördehoff, Bernhard
```

⁴ Wenn nichts anderes gesagt wird, heißt das Feld bzw. die Eigenschaft, nach der Sie gruppieren, in der späteren Auflistung so wie das Ausgangsfeld. Wenn Sie das nicht wünschen, können Sie das durch das Davorsetzen eines neuen Namens etwa mit Group By Lastname = adrElement.Nachname Into Kontaktliste = Group – an Ihre Wünsche anpassen. Anders als im Beispiel wäre hier Lastname die Eigenschaft, mit der Sie später die Nachnamen abfragen könnten.

Gruppieren von Listen aus mehreren Auflistungen

Group By eignet sich auch sehr gut dazu, mit Join kombinierte Listen zu gruppieren und auszuwerten. Stellen Sie sich vor, Sie möchten eine Liste mit Kontakten erstellen, mit der Sie über jeden Kontakt wieder auf eine Liste mit Artikeln zugreifen zu können, um auf diese Weise herauszufinden, welche Kunden welche Artikel gekauft hätten. Die entsprechende Abfrage und die anschließende Iteration durch die Ergebniselemente sähen dann folgendermaßen aus:

```
Sub GroupByJoinedCombined()
    Dim ergebnisliste = From adrElement In adrListe _
                        Join artElement In artListe _
                        On adrElement.ID Equals artElement.IDGekauftVon _
                        Group artElement.IDGekauftVon, artElement.ArtikelNummer, _
                        artElement.ArtikelName _
                        By artElement.IDGekauftVon, adrElement.Nachname, adrElement.Vorname _
                        Into Artikelliste = Group, AnzahlArtikel = Count() Order By Nachname

    For Each kontaktElement In ergebnisliste
        With kontaktElement

            Console.WriteLine(.IDGekauftVon & ": " & .Nachname & .Vorname)
            For Each artikel In .Artikelliste
                With artikel
                    Console.WriteLine(" " & .ArtikelNummer & ": " & .ArtikelName)
                End With
            Next
            Console.WriteLine()
        End With
    Next
End Sub
```

Hier sehen Sie eine Zusammenfassung dessen, was wir in den letzten Abschnitten kennen gelernt haben. Die Abfrage beginnt mit einem Join und vereint Artikel und Kundenliste zu einer flachen Auflistung, die sowohl die Namen als auch die Artikel für jeden Namen beinhaltet. Und dann wird gruppiert: Anders als beim ersten Gruppierungsbeispiel, in dem alle Elemente in der untergeordneten Liste einbezogen werden, geben wir hier zwischen Group und By die Felder an, die in der inneren Auflistung als Eigenschaften erscheinen sollen, wir ändern also, ähnlich dem Select-Befehl, das Schema der inneren Auflistung. Die Elemente, die wir anschließend hinter dem By angeben, sind die, nach denen gruppiert wird, und die damit auch in der äußeren Auflistung verfügbar sind. Das Ergebnis entspricht dann unseren Erwartungen:

```
21: Wördehoff, Theo
4-444: The Da Vinci Code
2-424: 24 - Season 6 [UK Import - Damn It!]
2-134: Abgefahren - Mit Vollgas in die Liebe
3-534: Mitten ins Herz
3-333: Der Schwarm
3-537: Visual Basic 2005 - Das Entwicklerbuch
4-444: Harry Potter und die Heiligtümer des Todes
5-554: O.C., California - Die komplette zweite Staffel (7 DVDs)
2-134: Abgefahren - Mit Vollgas in die Liebe
7-321: Das Herz der Hölle
```

75: Würdehoff, Katrin

2-134: Abgefahren - Mit Vollgas in die Liebe
2-134: Abgefahren - Mit Vollgas in die Liebe
2-134: Abgefahren - Mit Vollgas in die Liebe
3-123: Das Vermächtnis der Tempelritter
2-134: Abgefahren - Mit Vollgas in die Liebe
1-234: Das Leben des Brian
3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
3-543: Microsoft Visual C# 2005 - Das Entwicklerbuch
9-423: Desperate Housewives - Staffel 2, Erster Teil
7-321: Das Herz der Hölle
5-506: Visual Basic 2008 - Das Entwicklerbuch
5-513: Microsoft SQL Server 2008 - Einführung in Konfiguration, Administration, Programmierung
9-646: Was diese Frau so alles treibt
5-506: Visual Basic 2008 - Das Entwicklerbuch
2-321: Die Rache der Zwerge
9-445: Die Tore der Welt
4-444: Harry Potter und die Heiligtümer des Todes

77: Würdehoff, Theo

7-321: Das Herz der Hölle
1-234: Das Leben des Brian
9-009: Die Wächter
3-123: Das Vermächtnis der Tempelritter
5-518: Visual Basic 2008 - Neue Technologien - Crashkurs

Group Join

Exakt das gleiche Ergebnis bekommen Sie, allerdings mit sehr viel weniger Aufwand, wenn Sie sich der Group Join-Klausel bedienen, die Join und Group By miteinander kombiniert – das folgende Beispiel zeigt, wie's geht:

```
Sub GroupJoin()  
    Dim ergebnisliste = From adrElement In adrListe _  
                        Group Join artElement In artListe _  
                        On adrElement.ID Equals artElement.IDGekauftVon Into Artikelliste = Group  
  
    For Each kontaktElement In ergebnisliste  
        With kontaktElement  
  
            Console.WriteLine(.adrElement.ID & ": " & _  
                              .adrElement.Nachname & ", " & _  
                              & .adrElement.Vorname)  
  
            For Each artikel In .Artikelliste  
                With artikel  
                    Console.WriteLine(" " & .Artikelnummer & ": " & .ArtikelName)  
                End With  
            Next  
            Console.WriteLine()  
        End With  
    Next  
End Sub
```

Aggregatfunktionen

Aggregatfunktionen sind Funktionen, die ein bestimmtes Feld einer Auflistung aggregieren – zum Beispiel aufsummieren, den Durchschnitt berechnen, das Maximum oder das Minimum ermitteln oder einfach nur zählen, wie viele Elemente in einer Auflistung vorhanden sind. Anders als bei »normalen« Abfragen werden beim Aggregieren also die Elemente der Auflistung einer Aggregatfunktion alle nacheinander übergeben, und diese Funktion arbeitet bzw. rechnet dann mit jedem einzelnen dieser Elemente, betrachtet sie aber als Menge.

HINWEIS LINQ-Abfragen fangen grundsätzlich mit der FROM-Klausel an – reines Aggregieren von Auflistungen sind allerdings die einzige Ausnahme. Wenn Sie eine reine Aggregation auf eine Auflistung ausführen wollen, leiten Sie die LINQ-Abfrage nicht mit From sondern mit der Klausel Aggregate ein – die folgenden Beispiele werden es gleich demonstrieren.

Lassen Sie uns ganz einfach beginnen. Die erste Demo ...

```
Sub AggregateDemo()
    Dim scheinbarerUmsatz = Aggregate artElement In artListe _
        Into Summe = Sum(artElement.Einzelpreis)
    Console.WriteLine("Scheinbar beträgt der Umsatz {0:#,##0.00} Euro", scheinbarerUmsatz)
```

... aggregiert unsere Artikelliste in eine Gesamtsumme und ermittelt so scheinbar, wie viel Gesamtumsatz mit den Artikeln erzielt wurde, etwa wie hier zu sehen:

```
Scheinbar beträgt der Umsatz 47.117,60 Euro
```

Doch diese Aussage stimmt wirklich nur scheinbar. Denn bei der Abfrage wurden nur die Artikelpreise in einer Decimal-Variable (scheinbarerUmsatz) summiert – die Anzahl eines Artikel blieb unberücksichtigt.

Dieses Manko behebt das direkt anschließende Beispiel ...

```
Dim gesamtUmsatz = Aggregate artElement In artListe _
    Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl _
    Into Summe = Sum(Artikelumsatz)
Console.WriteLine("Der Umsatz beträgt {0:#,##0.00} Euro", gesamtUmsatz)
```

... das mit einem sehr, sehr alten Bekannten – nämlich der Let-Klausel – eine neue Variable einführt, die das *Produkt* von Anzahl und Preis aufnimmt und anschließend als Ergebnis aufsummiert:

```
Der Umsatz beträgt 93.306,00 Euro
```

Zurückliefern mehrerer verschiedener Aggregationen

Wenn eine Aggregat-Abfrage nur eine einzige Aggregat-Funktion verwendet, dann ist das Abfrageergebnis eine einzige Variable, die dem Rückgabebetyp der Aggregatfunktion entspricht. Liefert beispielsweise die Sum-Aggregatfunktion ein Ergebnis vom Typ Decimal, dann ist das Abfrageergebnis ebenfalls vom Typ Decimal.

Sie können aber auch mehrere Aggregate innerhalb einer einzigen Abfrage verwenden, wie das dritte Beispiel zeigt:

```
Dim mehrereInfos = Aggregate artElement In artListe _
    Let Artikelumsatz = artElement.Einzelpreis * artElement.Anzahl _
    Into Summe = Sum(Artikelumsatz), _
        Minimalpreis = Min(artElement.Einzelpreis), _
        Maximalpreis = Max(artElement.Einzelpreis) _
Console.WriteLine("Der Umsatz beträgt {0:#,##0.00} Euro" & vbCrLf & _
    "Minimal- und Maximalpreis jeweils " & _
    "{1:#,##0.00} und {2:#,##0.00} Euro", _
    mehrereInfos.Summe, mehrereInfos.Minimalpreis, _
    mehrereInfos.Maximalpreis)
```

In diesem Beispiel werden »in einem Rutsch« Artikelpostensumme, Maximalpreis und Minimalpreis ermittelt; die Rückgabeargument ist in dem Fall vom Typ anonyme Klasse mit den drei Eigenschaften, die über die entsprechenden Variablen innerhalb der Abfrage (Summe, Minimalpreis und Maximalpreis) definiert wurden. Hätten Sie diese Variablendefinitionen weggelassen und stattdessen nur die Aggregatfunktionen angegeben, hätte der Compiler für die Eigenschaften des anonymen Rückgabebetypen Namen gewählt, die den Namen der Aggregatfunktionen entsprechen.

WICHTIG Anders als bei Standardabfragen werden reine Aggregatabfragen, wie Sie sie hier in den letzten drei Beispielen kennen gelernt haben, *nicht* verzögert sondern sofort ausgeführt.

Kombinieren von gruppierten Abfragen und Aggregatfunktionen

Das Gruppieren von Abfragen eignet sich naturgemäß gut für den Einsatz von Aggregatfunktionen, denn beim Gruppieren von Datensätzen werden verschiedene Elemente zusammengefasst, auf die sich dann die unterschiedlichsten Aggregatfunktionen anwenden lassen.

Ein Beispiel soll auch das verdeutlichen: Angenommen, Sie möchten herausfinden, welcher Kunde durch seine Käufe wie viel Umsatz produziert hat. In diesem Fall würden Sie die Abfrage nach Artikeln gruppieren, und zwar nach dem Feld `IDGekauftVon` – denn dieses Feld repräsentiert »wer hat gekauft«. Damit in der Ergebnisliste nicht nur eine nichtssagende Nummer sondern ein greifbarer Vor- sowie ein Nachname zu finden sind, verknüpfen Sie Artikeltabelle und Kundentabelle mit einer `Join`-Abfrage oder – noch besser – fassen Gruppierung und Zusammenfassung der Tabellen mit einem `Group Join` zusammen. In die `Group Join`-Abfrage bauen Sie gleichzeitig noch die schon bekannte `Order By`-Klausel ein, mit der Sie die Ergebnisliste nach dem errechneten Umsatz sortieren – durch die Angabe von `Descending` als Schlüsselwort sogar absteigend, um eine `Top-Ranking-Umsatzliste` zu bekommen. Die eigentliche Aggregation der Umsätze erfolgt dann mit der hinter `Into` stehenden Klausel, in der ebenfalls bestimmt wird, mit welcher Eigenschaft später die innere Liste (`JoinedList`) im Bedarfsfall durchiteriert werden kann (und dabei dann einzelne `artListe`-Elemente vom Typ `Artikel` zurückliefert). Die Aggregatfunktion `Sum`, die dafür notwendig ist, summiert im Beispiel das Produkt von Menge und Articleinzelpreis auf, was – für jeden Kontakt gruppiert – exakt dem Umsatz jedes Kunden entspricht.

TIPP Wie ebenfalls im folgenden Beispiel zu sehen, verwendet die Abfrage die `Take`-Klausel, um die Ergebnisliste auf 10 Elemente zu beschränken. Es gibt weitere Klauseln, die Sie für solche Einschränkungen verwenden können, wie beispielsweise `Take While` oder `Skip` – deren Anwendungsweise ist ähnlich und über dieses Beispiel leicht ableitbar. Die Online-Hilfe verrät Ihnen zu diesen Klauseln mehr.

```
Sub JoinAggregateDemo()  
    Dim ergebnisListe = From adrElement In adrListe _  
                        Group Join artElement In artListe _  
                        On adrElement.ID Equals artElement.IDGekauftVon _  
                        Into _  
                        KundenUmsatz = Sum(artElement.Einzelpreis * artElement.Anzahl) _  
                        Order By KundenUmsatz Descending _  
                        Take 10  
  
    For Each item In ergebnisListe  
        With item.adrElement  
            Console.WriteLine(.ID & ": " & .Nachname & ", " _  
                              & .Vorname & " --- Umsatz:" _  
                              & item.KundenUmsatz.ToString("#,##0.00") & _  
                              " Euro")  
        End With  
    Next  
End Sub
```

Wenn Sie dieses Beispiel ausführen, produziert es in etwa die folgenden Zeilen:

```
3: Braun, Franz --- Umsatz:2.358,00 Euro  
34: Plenge, Franz --- Umsatz:2.287,80 Euro  
91: Jungemann, Katrin --- Umsatz:2.183,05 Euro  
94: Tinoco, Anne --- Umsatz:2.128,00 Euro  
100: Hollmann, Gareth --- Umsatz:2.108,15 Euro  
18: Westermann, Anja --- Umsatz:2.098,05 Euro  
66: Hörstmann, Katrin --- Umsatz:1.923,00 Euro  
54: Albrecht, Uta --- Umsatz:1.843,40 Euro  
97: Hollmann, Daja --- Umsatz:1.798,50 Euro  
27: Westermann, Alfred --- Umsatz:1.788,15 Euro
```