Kapitel 2

Einführung in LINQ

In diesem Kapitel:

Das bereits unter .NET 3.5 eingeführte LINQ (*Language Integrated Query*) ähnelt SQL, ist allerdings keine eigenständige Sprache, sondern eine grundlegend neue Technologie, welche das Implementieren aller Arten von Datenzugriffen vereinfacht und auf eine einheitliche Grundlage stellt, ohne an eine bestimmte Architektur gebunden zu sein.

Die LINQ-Philosophie

Weil das objektorientierte Paradigma das derzeit dominierende Software-Modell ist, müssen die Entwickler viel Zeit damit verbringen, um dieses Modell mit anderen Systemen, speziell relationale Datenbanken und XML, zu verbinden. Hier eilt uns LINQ zu Hilfe, indem es Transparenz und Produktivität der datenbezogenen Programmierung deutlich verbessert.

Aber es geht nicht nur um die Effektivität, sondern auch um die Qualität der Software-Entwicklung, denn das Schreiben von monotonen und fehleranfälligen Anpassungscode birgt die Gefahr von Instabilitäten in sich oder kann zur Herabsetzung der Performance führen.

Natürlich gab und gibt es neben LINQ bereits andere Lösungen. So könnten wir beispielsweise einen Codegenerator oder eines der verschiedenen objektrelationalen Mapping-Tools von Drittanbietern verwenden. Leider sind diese Tools alles andere als perfekt. Beispielsweise sind sie nur für den Datenbankzugriff geeignet und nicht für andere Datenquellen wie XML-Dokumente. Außerdem kann Microsoft etwas, was andere Anbieter nicht können, nämlich den Datenzugriff direkt in die Sprachen Visual Basic und C# integrieren.

OOP-Modell versus relationales Modell

Nehmen wir das objektorientierte und das relationale Modell, dann existiert der Widerspruch zwischen ihnen auf verschiedenen Ebenen:

- **Relationale Datenbanken und objektorientierte Sprachen verwenden nicht dieselben primitiven Datentypen.**
  Beispielsweise haben Strings in Datenbanken gewöhnlich eine begrenzte Länge, was in VB.NET oder in C# nicht der Fall ist. Das kann zum Problem werden, wenn Sie einen String mit 150-Zeichen in einem Tabellenfeld mit nur 100 Zeichen speichern wollen. Viele Datenbanken haben keinen booleschen Typ, während wir in Programmiersprachen oft *True*/*False* Werte verwenden.
- **OOP und relationale Theorien haben verschiedene Datenmodelle.**
  Aus Performancegründen und wegen ihres Wesens müssen relationale Datenbanken normalisiert werden. Normalisierung ist ein Prozess, bei dem Redundanzen eliminiert und Daten effektiv organisiert werden. Weiterhin wird das Potenzial für Anomalien während der Datenoperationen reduziert, auch die Datenkonsistenz wird verbessert. Normalisierung resultiert in einer Organisation der Daten die spezifisch für das relationale Datenmodell ist. Das verhindert ein direktes Mapping der Tabellen und Records mit Objekten und Auflistungen. Relationale Datenbanken werden in Tabellen und Beziehungen normalisiert, während Objekte Vererbung, Komposition und komplexe Referenzhierarchien verwenden. Ein grundsätzliches Problem existiert, weil relationale Datenbanken keine Konzepte wie das der Vererbung besitzen: das Mapping einer Klassenhierarchie an eine relationale Datenbank erfordert meist einige mehr oder weniger ausgefeilte Workarounds bzw. »Tricks«.

- **Programmiermodelle**
  In SQL schreiben Sie Abfragen und bewegen sich somit auf einer höheren Ebene der Deklaration, um auszudrücken, an welcher Datenmenge Sie interessiert sind. In allgemeinen Programmiersprachen wie C# oder VB müssen Sie hingegen Schleifenanweisungen, If-Statements usw. schreiben.
- **Kapselung**
  Objekte sind selbstenthaltend/selbstbeschreibend und enthalten sowohl Daten als auch Verhalten. In relationalen Datenbanken hingegen sind Code und Daten sauber voneinander getrennt.

HINWEIS Objektrelationales Mapping (ORM) ist die Brücke zwischen objektorientierten Sprachen und relationalen Datenbanken.

ORM kann als der Akt bezeichnet werden, bei welchem festgelegt wird, wie Objekte und ihre Beziehungen in einem permanenten Datenspeicher abgelegt (persistiert) werden, in diesem Fall in einer relationalen Datenbank.

Besonderheiten beim ORM

Konzepte wie Vererbung oder Komposition werden von relationalen Datenbanken nicht direkt unterstützt, d.h., die Daten können nicht auf gleiche Weise in beiden Modellen repräsentiert werden. Wie Sie am folgenden Beispiel sehen, können verschiedene Objekte und Typen an eine einzige Tabelle gemappt werden.

BEISPIEL

Die folgende Abbildung zeigt ein Objektmodell und ein entsprechendes relationales Modell. Wie man leicht erkennt, ist trotz der Einfachheit beider Modelle das Mapping wegen der Unterschiede zwischen beiden Paradigmen nicht ganz trivial.

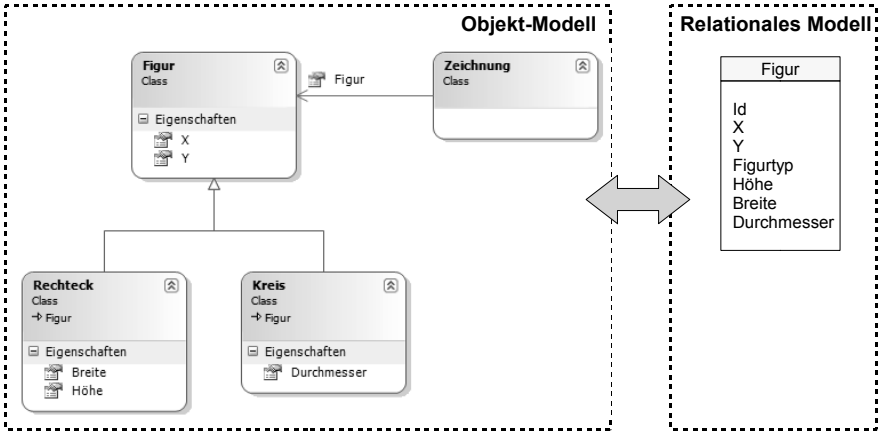

Abbildung 2.1 Objektmodell und relationales Modell

Auch wenn wir ein Objektmodell in einer neuen relationalen Datenbank speichern wollten, könnten wir kein direktes Mapping verwenden. Aus Performancegründen und um Duplikate zu vermeiden ist es im aktuellen Beispiel viel besser, wenn wir nur eine einzige Tabelle verwenden. Als Konsequenz können aber

dann die von der Datenbanktabelle kommenden Daten nicht genutzt werden, ohne dass der Objektgraph im Arbeitsspeicher aktualisiert wird. Wie Sie sehen, bedeutet ein Gewinn auf der einen Seite einen Verlust auf der anderen.

Wir könnten zwar ein Datenbankschema oder ein Objektmodell entwerfen welches diese Fehlanpassung zwischen beiden Welten reduziert, wir könnten diese aber niemals, wegen der beiden Paradigmen innewohnenden Unterschiede, beseitigen.

Meist haben wir nicht einmal diese Wahl, da das Datenbankschema bereits definiert ist. Ein anderes Mal müssen wir mit Objekten arbeiten, die jemand anderes entwickelt hat.

Das komplexe Problem der Integration von Datenquellen in Programme umfasst mehr als nur das einfache Lesen und Schreiben von bzw. in die Datenquelle. Wenn wir mit einer objektorientierten Sprache programmieren, wollen wir dass unsere Anwendung ein Objektmodell benutzt, welches eine konzeptionelle Repräsentation der Geschäftsdomäne darstellt, anstatt direkt an die relationale Struktur gebunden zu sein. Das Problem dabei ist, dass an bestimmten Stellen das Objektmodell und das relationale Modell zusammenarbeiten müssen. Das ist nicht leicht, weil objektorientierte Programmiersprachen und .NET Entity-Klassen Geschäftsregeln, komplexe Beziehungen und Vererbung umfassen, während eine relationale Datenquelle Tabellen, Zeilen, Spalten, Primär- und Fremdschlüssel usw. enthält.

Ein erstes LINQ-Beispiel

Ehe wir detailliert die hinter LINQ stehenden Konzepte beleuchten, wollen wir Ihnen an einem typischen Beispiel einen ersten Eindruck von den neuen Sprachkonstrukten vermitteln, wie sie ab VB 9.0 zur Verfügung stehen.

Klassischer Datenbankzugriff

Das in der .NET-Klassenbibliothek enthaltene ADO.NET (siehe Kapitel 3) stellt bereits eine umfangreiche API für den Zugriff auf relationale Datenbanken zur Verfügung und ermöglicht die Repräsentation relationaler Daten im Speicher. Das Problem mit diesen Klassen (z.B. *SqlConnection*, *SqlCommand*, *SqlReader*, *DataSet, DataTable*) ist aber, dass sie den Entwickler dazu zwingen, explizit mit Tabellen, Records und Spalten zu arbeiten, während eine moderne Sprachen wie VB.NET auf dem objektorientierte Paradigma basiert. Das folgende Beispiel zeigt eine typische Anwendung im herkömmlichen Sinn.

BEISPIEL

Datenbankzugriff unter dem klassischen .NET

```
...
Dim connection As New SqlConnection("...")
Dim command As SqlCommand = connection.CreateCommand()
```

SQL-Abfrage als String:

```
command.CommandText = "SELECT CompanyName, Country FROM Customers  WHERE City = @City"
```

Lose gebundene Parameter:

```
command.Parameters.AddWithValue("@City", "London")
```

```
    connection.Open()
    Dim reader As SqlDataReader = command.ExecuteReader()
    While reader.Read()
```

Lose typisierte Spalten:

```
        Dim name As String = reader.GetString(0)
        Dim land As String = reader.GetString(1)
```

Ausgabe:

```
        ListBox1.Items.Add(name & "  " & land)
    End While
    reader.Close()
    connection.Close()
    ...
```

Folgende Einschränkungen bzw. Fragen ergeben sich aus obigem Code:

- Alle SQL-Befehle sind als Zeichenketten notiert, sie unterliegen deshalb keinerlei Prüfungen durch den Compiler. Was also ist, wenn der String eine ungültige SQL-Abfrage enthält? Was passiert, wenn eine Spalte in der Datenbank umbenannt wurde?
- Dasselbe trifft auch auf die Parameter und auf die Ergebnismenge zu – sie sind nur lose definiert. Haben die Spalten den richtigen Typ? Haben wir die korrekte Anzahl von Parametern übergeben? Entsprechen die Parameter in der Abfrage exakt den Parameterdeklarationen?
- Die von uns verwendeten Klassen sind nur für den SQL Server bestimmt und können nicht von einem anderen Datenbankserver benutzt werden.

Das grundsätzliche Problem beim Zugriff auf relationale Datenbanken ist, dass sich eine tiefe Kluft zwischen Ihrer Programmiersprache und der Datenbank auftut, d.h., zwischen der relationalen und der objektorientierten Sicht gibt es Differenzen, die nicht einfach zu überbrücken sind. Zwar wurden viele mehr oder weniger erfolgreiche Versuche zur Einführung objektorientierter Datenbanken unternommen, die näher an objektorientierten Plattformen und wichtigen Programmiersprachen wie VB und C# angesiedelt sind, trotzdem ist diesen Datenbanken der große Durchbruch bis heute versagt geblieben und relationale Datenbanken sind nach wie vor weit verbreitet. D.h., Sie als Programmierer müssen nach wie vor mit dem Datenzugriff kämpfen.

Hierin also liegt die ursprüngliche Motivation, die zur Entwicklung von LINQ führte, nämlich die Überwindung des Widerspruchs zwischen den objektorientierten .NET-Programmiersprachen und den nach wie vor den Markt dominierenden relationalen Datenbanken.

Datenbankzugriff unter LINQ

Mit LINQ beabsichtigt Microsoft eine Lösung des Problems des objekt-relationalen Mapping (ORM), genauso wie die Vereinfachung der Interaktion zwischen Objekten und Datenquellen. Das folgende Beispiel zeigt eine Realisierung des Vorgängerbeispiels mit den neuen LINQ-Sprachkonstrukten (fett) und dürfte einen ersten Eindruck über die Vorteile dieser Technologie vermitteln.

BEISPIEL

Einfache LINQ-Abfrage (Prinzip)

Datenkontext instanziieren:

```
Dim DB As New MyDataContext()
```

LINQ to SQL-Abfrage formulieren:

```
Dim customers = From cust In DB.Customers
                Where cust.City = "London"
                Select cust.CompanyName, cust.Country
```

Durchlaufen der *customers*-Collection und Ausgabe:

```
For Each cust In customers
    ListBox1.Items.Add(cust.CompanyName & " " & cust.Country)
Next
```

Dieser Code überrascht durch seine Transparenz und Kürze, leistet aber das Gleiche wie das in klassischer ADO.NET-Technologie realisierte Vorgängerbeispiel. Obwohl eine Erklärung der neuen Sprachkonstrukte erst an späterer Stelle erfolgt, dürfte der Code bereits jetzt weitgehend selbsterklärend sein. Mehr noch: In LINQ-Abfragen können die Daten im Arbeitsspeicher, in einer Datenbank, in einem XML Dokument oder an einer anderen Stelle sein, die Syntax bleibt dieselbe. Wie wir noch sehen werden, kann dank der Erweiterbarkeit von LINQ diese Art von Abfragen auch mit mehreren Datentypen und verschiedenen Datenquellen verwendet werden.

Damit zeigt das Beispiel aber nur die Spitze des Eisbergs, denn LINQ hat sich aus seiner ursprünglichen Zweckbestimmung (Zugriff auf relationale Datenbanken) heraus zu einer allgemeinen Sammlung von Abfragewerkzeugen, die in eine Sprache integriert werden können, weiter entwickelt. Diese Tools werden verwendet um auf Daten zuzugreifen, die von In-Memory Objekten (LINQ to Objects), Datenbanken (LINQ to SQL), XML Dokumenten (LINQ to XML), dem Dateisystem oder von irgendeiner anderen Quelle kommen.

Der Weg zu LINQ

Wie wir bereits wissen, erfüllte Microsoft mit LINQ den Wunsch vieler Entwickler nach universell einsetzbaren Datenabfragemethoden, die SQL-Abfragen ähneln. Microsoft hatte bis dahin noch keine Lösung für das objektrelationale Daten-Mapping (ORM) und mit LINQ bot sich für MS die Gelegenheit, sowohl Mapping als auch Abfragemechanismen in seine .NET-Programmiersprachen zu integrieren.

Wie war es in der Zeit vor LINQ?

Damals mussten Sie sich mit verschiedenen Sprachen, wie SQL, XML oder XPath, und verschiedenen Technologien und APIs, wie ADO.NET oder *System.Xml*, in jeder Anwendung die mit allgemeinen Sprachen wie VB oder C# geschrieben wurden, herumplagen und waren ausschließlich auf die unterschiedlichen Datenzugriffsmodelle angewiesen bzw. mussten dafür eigene APIs entwickeln. So haben Sie für die Abfrage von Datenbanken üblicherweise SQL verwendet. Für den Zugriff auf XML-Dokumente benutzten Sie das DOM (*Document Object Model*) oder XQuery. Mit Schleifenanweisungen haben Sie Arrays durchsucht, oder Sie

haben selbst spezielle Algorithmen geschrieben, um sich durch Objektbäume zu hangeln oder auf andere Arten von Daten wie Excel-Tabellen, E-Mails oder die Registrierdatenbank zuzugreifen. Unterm Strich war es also bislang so, dass verschiedene Datenquellen verschiedene Programmiermodelle erforderlich machten.

Ein weiteres Problem sind die verschiedenen Datentypen, mit denen Sie zu kämpfen haben, wenn ein bestimmtes Datenmodell nicht an die jeweilige Sprache gebunden ist. Das kann zu Anpassungsschwierigkeiten zwischen Daten und Code führen. LINQ stellt die bislang vermisste direkte Verbindung zwischen den unterschiedlichen Datenquellen und verschiedenen (.NET)-Programmiersprachen her und vereinheitlicht den Datenzugriff auf beliebige Datenquellen. Es erlaubt Abfrage- und Schreiboperationen, ähnlich wie bei SQL Anweisungen für Datenbanken. Durch zahlreiche Spracherweiterungen integriert LINQ diese Abfragen direkt in .NET-Sprachen wie Visual Basic und C#.

So sind beispielsweise der objektorientierte Zugriff auf XML oder das Mixen relationaler Daten mit XML einige der Aufgaben die LINQ vereinfacht.

Ein zentraler Aspekt ist weiterhin, dass LINQ die Zusammenarbeit mit beliebigen Typen von Objekten oder Datenquellen ermöglicht und dazu ein konsistentes Programmiermodell bereitstellt. Syntax und Konzepte für den Datenzugriff sind dieselben. Wenn Sie also wissen wie LINQ mit einem Array oder einer Collection funktioniert, dann kennen Sie auch die grundlegenden Konzepte für die Anwendung von LINQ auf Datenbanken oder XML-Dateien.

Ein weiterer Vorteil von LINQ ist, dass Sie damit in einer streng typisierten Welt arbeiten. Beim Entwurf erhalten Sie Hinweise von der IntelliSense und Ihre Abfragen werden bereits beim Kompilieren geprüft.

HINWEIS LINQ wird einige Aspekte Ihres Umgangs mit Daten in Anwendungen und Komponenten grundlegend verändern. Sie werden merken, dass LINQ ein wesentlicher Schritt in Richtung deklaratives Programmieren darstellt. Sie werden sich sehr bald darüber wundern, warum Sie früher so viele Zeilen Code geschrieben haben.

Wichtige LINQ-Features

LINQ ermöglicht durch das Schreiben von Abfragemethoden den Zugriff auf jede beliebige Art von Datenquelle.

Konkret stellen sich die Vorteile von LINQ wie folgt dar:

- Erweiterung der Sprachen Visual Basic und C#
- streng typisiert
- Queries werden zur Entwurfszeit geprüft, nicht erst zur Laufzeit
- Intellisense-Unterstützung in Visual Studio
- keine CLR-Erweiterung notwendig

Die Funktionsweise:

- LINQ-Ausdrücke werden vom Compiler in Extension Methods (Erweiterungsmethoden) und Lambda Expressions übersetzt
- Durch die Lambda Expressions steht der LINQ-Ausdruck zur Laufzeit als Expression Tree (Abfragebaum) zur Verfügung
- Die Expression Trees werden je nach LINQ-Variante in eine andere Darstellung (wie z.B. SQL) übersetzt

LINQ Ausdrücke können auf folgende Typen angewendet werden:

- *IEnumerable* (LINQ to Objects)
- *IQueryable(Of T)* (LINQ to Entities, ... to SQL, ...)

Trotz all dieser euphorisch stimmenden LINQ-Features sollte eines nicht vergessen werden: SQL ist ein weit verbreiteter Standard und kann auch in Zukunft nicht generell durch LINQ ersetzt werden. Allerdings ist der deklarative Charakter der LINQ-Syntax unbestritten ein Meilenstein bei der Entwicklung der großen Programmiersprachen.

Die LINQ-Architektur

Die Abbildung 2.2 soll die grundsätzliche Architektur von LINQ verdeutlichen.

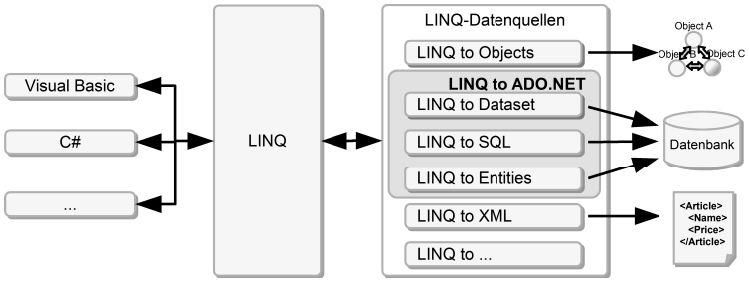

Abbildung 2.2 Die LINQ Architektur

Je nach Standort des Betrachters besteht LINQ einerseits aus einer Menge von Werkzeugen zur Arbeit mit Daten, was in den verschiedenen LINQ-Implementierungen (LINQ to Objects, LINQ to DataSets, LINQ to SQL, LINQ to Entities und LINQ to XML) zum Ausdruck kommt. Andererseits besteht LINQ aus einer Menge von Spracherweiterungen (momentan für VB und C#).

LINQ-Implementationen

LINQ bietet zahlreiche Varianten für den Zugriff auf verschiedenste Arten von Daten. Diese sind in den verschiedenen LINQ-Implementationen (auch als »LINQ Flavours«, d.h. »Geschmacksrichtungen« bezeichnet) enthalten. Folgende LINQ-Provider werden als Bestandteil des .NET Frameworks bereitgestellt:

- LINQ to Objects (arbeitet mit Collections die *IEnumerable* implementieren)
- LINQ to XML (Zugriff auf XML Strukturen)
- LINQ to SQL (Zugriff auf SQL Datenbanken)
- LINQ to DataSet (arbeitet auf Basis von DataSets) und
- LINQ to Entities (verwendet das ADO.NET Entity Framework als ORM)

Diese LINQ-Provider/Implementationen bilden eine Familie von Tools, die einzeln für bestimmte Aufgaben eingesetzt oder aber auch für leistungsfähige Lösungen mit einem Mix aus Objekten, XML und relationalen Daten miteinander kombiniert werden können.

HINWEIS Nochmals sei hier betont, dass LINQ eine offene Technologie ist, der jederzeit neue Provider hinzugefügt werden können! Die im .NET Framework 3.5 enthaltenen Implementationen bilden lediglich eine Basis, die eine Menge von Grundbausteinen (Abfrageoperatoren, Abfrageausdrücke, Abfragebäume) bereitstellt.

Die neuen Sprachfeatures

Für die Einbindung von LINQ in VB mussten mehrere neue Sprachkonstrukte eingeführt werden, die wir Ihnen im Folgenden vorstellen möchten. Teilweise bauen diese auf den bereits unter .NET 2.0 eingeführten Neuerungen (generische Typen etc.) auf.

Typinferenz

Unter Typinferenz versteht man ein Sprachmerkmal welches es erlaubt, dass der Datentyp lokaler Variablen bei der Deklaration vom Compiler automatisch ermittelt wird, ohne dass explizit der Typ angegeben werden muss. Wie wir später noch sehen werden, erweist sich dieses Feature vor allem für anonyme Typen als praktisch bzw. notwendig.

Als Ersatz für einen konkreten Typ wird in VB das Schlüsselwort *Dim* verwendet, wobei man auf das *As* verzichtet.

Mancher mag denken, dass es sich hier um dasselbe Verhalten wie bei *Option Strict Off* handelt, tatsächlich aber erhalten Sie streng typisierte Variablen.

BEISPIEL

Die Initialisierung der Variablen *a* wird vom Compiler ausgewertet und der Typ aufgrund des Wertes 35 auf *Integer* festgelegt.

```
Dim a = 35
```

Obige Zeile ist semantisch identisch mit folgendem Ausdruck:

```
Dim a As Integer = 35
```

Der Datentyp wird einmalig bei der ersten Deklaration der Variablen vom Compiler festgelegt und kann danach nicht mehr verändert werden!

BEISPIEL

Da die Variable *b* vom Compiler als *Integer* festgelegt wurde, kann ihr später kein *Double*-Wert zugewiesen werden.

```
Dim b = 7
b = 12.3
MessageBox.Show(b.ToString)          ' zeigt 12
```

BEISPIEL

Verschiedene implizite lokale Variablendeklarationen

```
Dim txt = "abc"                      ' Typ : String
Dim pi = 3.1415                      ' Typ Double
For i = 1 To 10                      ' Typ von i: Integer
    ...
```

Obige Beispiele sind semantisch äquivalent zur »klassischen« Deklaration bei welcher der Datentyp explizit angegeben wird.

HINWEIS Typinferenz ist nicht gleichbedeutend mit dem Zuweisen des *Object*-Datentyps, wie folgendes Beispiel belegt.

BEISPIEL

Zuweisen einer per Typinferenz deklarierten Integer-Variablen zu einer Variablen vom Typ *Object*

```
Option Strict On
...
Dim a = 5                               ' Deklarieren von a als Integer
Dim b As Object = 3                     ' Boxing von Integer in Object
Dim c As Integer= a                     ' kein Casting, kein Unboxing
Dim d As Integer = CType(b, Integer)    ' Casting und Unboxing sind notwendig
```

HINWEIS Wie bereits erwähnt, ist die Typinferenz nur im lokalen Gültigkeitsbereich zulässig, d.h., sie darf nicht für Membervariablen und Parameter benutzt werden.

BEISPIEL

Einige zulässige implizite Variablendeklarationen

```
    Public Sub test(d As Decimal)
        Dim x = 5.3             ' Double
        Dim y = x               ' Double
        Dim r = x / y           ' Double
        Dim s = "test"          ' String
        Dim w = d               ' Decimal
    End Sub
```

BEISPIEL

Nicht zulässige implizite Variablendeklarationen

```
Option Strict On

Class Form1
    Dim a = 0                           ' Fehler, da nicht lokal

    Public Sub test1(Dim x)             ' Fehler, da Typ in Parameterliste erwartet wird
        ...
    End Sub

    Public Function test2()             '  Fehler, da Funktionstyp erwartet wird
        Return 5
    End Function

End Class
```

HINWEIS Obwohl durch verwenden von Typinferenz das Schreiben von Code vereinfacht wird, kann die Lesbarkeit darunter erheblich leiden (besonders bei mehreren Methodenüberladungen).

Nullable-Typen

Der Compiler kann durch ein der Typdeklaration nachgestelltes Fragezeichen (?) einen Wertetyp in eine generische *System.Nullable(Of T As Structure)*-Struktur verpacken.

BEISPIEL

Einige Deklarationen von Nullable-Typen

```
Dim i As Integer? = 10
Dim j As Integer? = Nothing
Dim k As Integer? = i + j              ' Nothing
```

Von Nutzen sind Nullable-Typen besonders dann, wenn Daten aus einer relationalen Datenbank gelesen bzw. dorthin zurückgeschrieben werden sollen.

Was sind Nullable Types?

Einer der Hauptunterschiede zwischen Wertetypen wie *Integer* oder *Structure* und Referenztypen wie *Form* oder *String* ist der, dass Referenztypen so genannte Null-Werte unterstützen. Eine Referenztyp-Variable kann also den Wert *Nothing* enthalten, d.h., die Variable referenziert im Moment keinen bestimmten Wert. Demgegenüber enthält eine Wertetyp-Variable immer einen Wert, auch wenn dieser, wie bei einer *Integer*-Variablen, den Wert 0 (null) hat. Falls Sie einer Wertetyp-Variablen *Nothing* zuweisen, wird diese auf ihren Default-Wert zurückgesetzt, bei einer *Integer*-Variablen wäre das 0 (null).

Die aktuelle CLR bietet keine Möglichkeit um festzustellen, ob einer *Integer*-Variablen ein Wert zugewiesen wurde, die Tatsache dass die Variable den Wert 0 (null) hat bedeutet noch lange nicht, dass dieser Wert auch zugewiesen wurde, denn es könnte sich ja genauso gut um den Default-Value handeln.

Leider gibt es Situationen, wo es wünschenswert wäre, dass auch Wertetypen einen Null-Zustand annehmen könnten. Das allgemeinste Beispiel für solch einen Fall wäre ein Typ, der Informationen aus einer Datenbank repräsentiert. Viele Datenbanken erlauben Spalten beliebiger Typen, dass diese einen Null-Wert erhalten können. Das bedeutet, »man hat dieser Spalte noch keinen Wert zugewiesen«.

BEISPIEL

Eine Personal-Datenbank kann einen Null-Wert für die Spalte *Gehalt* für pensionierte Angestellte zulassen, d.h., dass diese nicht länger ein Gehalt beziehen[1].

Die Tatsache, dass die CLR keine Nullwerte für Wertetypen unterstützt, kann in solch einer Situation peinlich sein und ist auch der Grund dafür, dass es den *System.Data.SqlTypes* Namespace gibt. Die darin enthaltenen Typen sind speziell für SQL-Anwendungen optimiert – wäre es aber nicht schön, wenn wir dieses Verhalten für alle Wertetypen hätten?

Die Antwort ist »ja« und die Unterstützung für generische Typen macht es möglich. Ein neuer generischer Typ *Nullable(Of T)* ermöglicht es auch Wertetypen, »nullable« zu sein.

[1] Einen mysteriösen Wert wie 0 Euro zuzuweisen wäre irreführend, da man meinen könnte, der Angestellte arbeite ohne Geld.

BEISPIEL

Ein an die Subroutine *PrintValue* übergebener Integer-Wert wird nur angezeigt, wenn ihm ein Wert zugewiesen wurde. Ansonsten erfolgt die Ausgabe »Null Wert«.

```
Sub PrintValue(i As Nullable(Of Integer))
    If i.HasValue Then
      Console.WriteLine(CInt(i))
    Else
      Console.WriteLine("Null Wert!")
    End If
End Sub
```

Objekt-Initialisierer

Durch Objekt-Initialisierer wird es möglich, ähnlich wie bei der Initialisierung von Attributen, elegant Felder und Eigenschaften einer Klasse oder Struktur auf Anfangswerte zu setzen. Somit können nun öffentliche Eigenschaften und Felder von Objekten ohne das explizite Vorhandensein eines passenden Konstruktors in beliebiger Reihenfolge initialisiert werden.

HINWEIS Wie wir noch sehen werden, ist diese Funktionalität notwendig um anonyme Typen zu initialisieren.

Das Initialisieren geschieht mit dem Schlüsselwort *With* und nachfolgenden geschweiften Klammern, in denen die einzelnen Felder/Eigenschaften des Objekts mit Werten belegt werden. Der Namen der Felder/Eigenschaften muss mit einem Punkt (.) beginnen.

BEISPIEL

Ausgangspunkt ist eine Klasse C*Kunde*:

```
Public Class CKunde
    Public Name As String
    Public PLZ As Integer
    Public Ort As String
End Class
```

Das Erzeugen und Initialisieren einer Instanz von *CKunde* bedarf keines speziellen Konstruktors:

```
Private kunde As New CKunde With {.Name = "Müller", .PLZ = 12345, .Ort = "Musterhausen" }
```

BEISPIEL

Verschachtelte Objektinitialisierung beim Erzeugen einer Instanz der Klasse *Rectangle*:

```
 Dim rect As New Rectangle With {
    .Location = New Point With {.X = 3, .Y = 7},
    .Size = New Size With {.Width = 19, .Height = 34}}
```

BEISPIEL

Initialisierung einer Collection aus Objekten der Klasse *CKunde*

```
Dim kunden() = {New CKunde With {.Name = "Müller", .PLZ = 12345, .Ort = "Musterhausen"},
```

```
                New CKunde With {.Name = "Meier", .PLZ = 2344, .Ort = "Walldorf"},
                New CKunde With {.Name = "Schulze", .PLZ = 32111, .Ort = "Biesdorf"}}
```

Auf ähnliche Weise wie bei Objektinitialisierern kann man auch zu Auflistungen, die *ICollection(Of T)* implementieren, elegant Elemente hinzufügen. Für die Elemente wird, entsprechend ihrer Reihenfolge, die Methode *ICollection(Of T).Add(element As T)* aufgerufen. Die aufgelisteten Elemente müssen natürlich vom Typ *T* sein oder es muss eine implizite Konvertierung zu *T* existieren.

BEISPIEL

Erzeugen und Initialisieren einer Auflistung.

```
Dim intList As New List(Of Integer) (New Integer() {0, 2, 2, 3, 6, 10, 15, 29, 44})
```

BEISPIEL

Diese Anweisung erzeugt einen Fehler, denn *Double* kann nicht nach *Integer* konvertiert werden.

```
Option Strict On
...
Dim intList As New List(Of Integer) (New Integer() {0, 2, 3, 5, 7.49})     ' Fehler!
```

Anonyme Typen

Darunter verstehen wir einfache namenlose Klassen, die vom Compiler automatisch erzeugt werden und die nur über Eigenschaften und dazugehörige private Felder verfügen. »Namenlos« bedeutet, dass uns der Name der Klasse nicht bekannt ist und man deshalb keinen direkten Zugriff auf die Klasse hat. Lediglich eine Instanz steht zur Verfügung, die man ausschließlich lokal, d.h. im Bereich der Deklaration, verwenden kann.

Das Deklarieren anonymer Typen erfolgt mittels eines anonymen Objekt-Initialisierers, d.h., man lässt beim Initialisieren einfach den Klassennamen weg. Der Compiler erzeugt die anonyme Klasse anhand der Eigenschaften im Objekt-Initialisierer und anhand des jeweiligen Typs der zugewiesenen Werte.

BEISPIEL

Eine Objektvariable *person* wird aus einer anonymen Klasse instanziiert.

```
Dim person = New With {.Vorname = "Maxhelm", .Nachname = "Müller", .Alter = 53}
```

Der Compiler generiert hierfür intern den MSIL-Code, der der folgenden Klasse entspricht:

```
Friend Class ???????
   Private _vorname As String
   Private _nachname As String
   Private _alter As Integer

   Public Property Vorname() As String
     Get
       Return _vorname
     End Get
     Set(value As String)
       _vorname = value
```

```
    End Set
  End Property

  Public Property Nachname() As String
    Get
      Return _nachname
    End Get
    Set(value As String)
      _nachname = value
    End Set
  End Property

  Public Property Alter() As Integer
    Get
      Return _alter
    End Get
    Set(value As Integer)
      _alter = value
    End Set
  End Property
End Class
```

Sobald eine weitere anonyme Klasse deklariert wird, bei der im Objekt-Initialisierer Eigenschaften mit dem gleichen Namen, Typ und in der gleichen Reihenfolge wie bei einer anderen bereits vorhandenen anonymen Klassen angegeben sind, verwendet der Compiler die gleiche anonyme Klasse, und es sind untereinander Zuweisungen möglich.

BEISPIEL

Da Name, Typ und Reihenfolge der Eigenschaften im Objekt-Initialisierer bei *person* (siehe oben) und *kunde* identisch sind, ist ein direktes Zuweisen möglich.

```
Dim kunde = New With {.Vorname = "Siegbast", .Nachname = "Krause", .Alter = 29}
kunde = person
```

Erweiterungsmethoden

Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Visual Basic 9.0 führte eine neue Syntax ein, die das direkte Hinzufügen neuer Methoden zu einer bereits vorhandenen Klasse erlaubt. Mit anderen Worten: Mit Erweiterungsmethoden können Sie einem Datentyp oder einer Schnittstelle Methoden außerhalb der Definition hinzufügen.

In Visual Basic müssen sowohl die Erweiterungsmethode als auch das Modul, welches die Erweiterungsmethode enthält, mit dem Attribut *System.Runtime.CompilerServices.Extension* versehen werden.

BEISPIEL

Die Klasse *System.Int32* wird um die Methoden *mult()* und *abs()* erweitert.

```
Imports System.Runtime.CompilerServices
<Extension()>
Public Module IntExtension
    <Extension()>
```

```
    Public Function mult(i As Integer, faktor As Integer) As Integer
        Return i * faktor
    End Function

    <Extension()>
    Public Function abs(i As Integer) As Integer
        If i < 0 Then i = -1 * i
        Return i
    End Function
End Module
```

Der Test:

```
    Private Sub Button1_Click(sender As Object, e As System.EventArgs) Handles Button1.Click
        Dim zahl As Integer = -95
        TextBox1.Text = zahl.mult(7).ToString              ' -665
        TextBox2.Text = zahl.abs.ToString                  '   95
    End Sub
```

In diesem Beispiel kann man nun die Erweiterungsmethoden *mult* und *abs* für jede Integer-Variable so nutzen, als wären diese Methoden direkt in der Basisklasse *System.Int32* als Instanzenmethoden implementiert.

HINWEIS Falls in *System.Int32* bereits eine *abs*-Methode mit der gleichen Signatur wie die gleichnamige Erweiterungsmethode existieren würde, so hätte die in *System.Int32* bereits vorhandene Methode Vorrang vor der Erweiterungsmethode.

Erweiterungsmethoden ermöglichen es Ihnen, einem vorhandenen Typ neue Methoden hinzuzufügen, ohne den Typ tatsächlich zu ändern. Die Standardabfrageoperatoren in LINQ stellen eine Reihe von Erweiterungsmethoden dar, die Abfragefunktionen für jeden Typ bieten, der *IEnumerable(Of T)* implementiert.

BEISPIEL

Durch die folgende Erweiterungsmethode wird der *String*-Klasse eine *Print*-Methode hinzugefügt.

```
<Extension()>
Public Sub Print(s As String)
    Console.WriteLine(s)
End Sub
```

Die Methode wird wie jede normale Instanzenmethode von *String* aufgerufen:

```
Dim msg As String = "Hallo"
msg.Print()
```

Lambda-Ausdrücke

Die Lambda-Ausdrücke (*Lambda Expressions*) gehörten zweifelsfrei mit zu den spektakulärsten sprachlichen Neuerungen von VB 9.0. Die Syntax basiert auf dem Schlüsselwort *Function*:

```
Function(Inputparameter) Expression
```

Die *Inputparameter* werden im *Expression* ausgewertet und liefern so gewissermaßen den Rückgabewert des Lambda-Ausdrucks.

Ein Lambda-Ausdruck ist quasi eine namenlose (anonyme) Funktion, von der ein einzelner Wert berechnet und zurückgegeben wird. Im Gegensatz zu benannten Funktionen kann ein Lambda-Ausdruck gleichzeitig definiert und ausgeführt werden.

BEISPIEL

Eine Methode zum Multiplizieren von zwei Gleitkommazahlen wird mittels Lambda-Ausdruck definiert.

```
Public Delegate Function opDeleg(x As Double, y As Double) As Double
```

Methode als Lambda-Ausdruck zuweisen:

```
Dim multDlg As opDeleg = Function(x As Double, y As Double) x * y
```

Der Test:

```
TextBox1.Text = multDlg(5.5, 4.3).ToString            ' 23.65
```

Einen kleinen Vorgeschmack auf den Einsatz von Lambda-Ausdrücken im Zusammenspiel mit LINQ-Ausdrücken soll das folgende Beispiel liefern:

BEISPIEL

Definition einer Abfrage über eine (generische) Liste

```
Dim employees As New List(Of Employee)
Dim query = employees.FindAll(Function(c) c.City = "London")
```

HINWEIS In LINQ liegen vielen der Standardabfrageoperatoren Lambda-Ausdrücke zugrunde, diese werden vom Compiler erstellt, um Berechnungen zu erfassen, die in grundlegenden Abfragemethoden wie *Where, Select*, *Order By*, *Take While* usw. definiert sind.

Der Rückgabetyp eines Lambda-Ausdrucks wird durch den Typ des rechts stehenden Ausdrucks bestimmt. Folglich hat ein Lambda-Ausdruck mit nur einem Methodenaufruf den gleichen Rückgabetyp wie diese Methode.

BEISPIEL

Einige Lambda-Ausdrücke, die die Zuordnung des Rückgabetyps veranschaulichen sollen.

Rückgabetyp ist leer, da *Console.WriteLine()* nichts zurückliefert:

```
Function(j As Integer) Console.WriteLine(j.ToString)
```

Rückgabetyp *Integer*, da *j* und *k* vom Typ *Integer* sind:

```
Function(j As Integer, k As Integer) j * k
```

Rückgabetyp *Double*, da ein *Double (0.7)* zu einem *Integer* addiert wird und das Ergebnis *Double* ist:

```
Function(i As Integer) i + 0.7
```

Rückgabetyp *String*, da auf der rechten Seite ein *String* addiert wird:

```
Function(geb As Integer) "Alles Gute zum " & geb & ". !"
```

Natürlich müssen Anzahl der Parameter und deren jeweiliger Datentyp mit denen des Delegaten, für den der Lambda-Ausdruck angegeben wird, übereinstimmen. Allerdings kann bei Lambda-Ausdrücken auf die Angabe des Typs für die Parameter verzichtet werden, da diese vom Kontext her ableitbar sind. Dem jeweiligen Parameter des Lambda-Ausdrucks wird also automatisch der Typ des entsprechenden Parameters des Delegaten, für den der Lambda-Ausdruck angegeben wird, zugewiesen. Im folgenden Beispiel kann man dieses Konzept erkennen:

BEISPIEL

Parametertyperkennung in einem Lambda-Ausdruck

```
Public Class CPerson
   Public Name As String
   Public Alter As Integer
End Class
```

Schnell eine Instanz mittels Objektinitialisierer erzeugen:

```
Public person1 As New CPerson With {.Name = "Krause", .Alter = 45}
```

Einen Delegate-Typ definieren, der eine *CPerson*-Instanz als Parameter entgegennimmt und nichts zurückgibt (*Sub*):

```
Public Delegate Sub PDelegate(person As CPerson)
```

Wegen *PDelegate* ist der Parameter *p* vom Typ *CPerson*, der Rückgabetyp muss leer sein:

```
Dim dlg1 As PDelegate = Function(p) _
                        MessageBox.Show(p.Name & " ist " & p.Alter.ToString & " Jahre alt!")
```

Der Aufruf ist unspektakulär:

```
dlg1(person1)
```

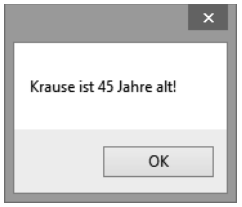

Abbildung 2.3 Ergebnis des Beispiels

Allgemein bleibt es die Entscheidung des Entwicklers, ob er benannte Methoden oder Lambda-Ausdrücke verwenden möchte. Lambda-Ausdrücke haben den Vorteil der einfachsten und kompaktesten Syntax. Wichtiger noch dürfte es aber sein, dass Lambda-Ausdrücke sowohl als Code als auch als Daten kompiliert werden, was ihre Verarbeitung zur Laufzeit, d.h. ihre Optimierung, Übersetzung und Bewertung, ermöglicht.

Abfragen mit LINQ to Objects

Bei *LINQ to Objects* handelt es sich um die allgemeinste und grundlegendste LINQ-Implementierung, welche auch die wichtigsten Bausteine für die übrigen LINQ-Implementierungen liefert. In einer SQL-ähnlichen Syntax können miteinander verknüpfte Collections/Auflistungen abgefragt werden, die über die *IEnumerable*-Schnittstelle verfügen. LINQ-Code kann grundsätzlich in so genannter *Query Expression Syntax* oder in *Extension Method Syntax* geschrieben werden, auch Mischformen sind möglich.

Grundlegende Syntax

Dazu gehört zunächst als wichtigster Standard die Angabe einer Quelle (*From*), das Festlegen der zurückzugebenden Daten (*Select),* das Filtern (*Where*) und das Sortieren (*Order By*). Hinzu kommt eine Fülle weiterer Operatoren, wie z.B. für das Gruppieren, Verknüpfen und Sammeln von Datensätzen usw.

Die LINQ-Abfrageoperatoren sind als Erweiterungsmethoden definiert und in der Regel auf beliebige Objekte, die *IEnumerable(Of T)* implementieren, anwendbar.

BEISPIEL

Gegeben sei die Auflistung:

```
Dim monate() As String = {"Januar", "Februar", "März", "April", "Mai", "Juni",
                          "Juli", "August", "September", "Oktober", "November", "Dezember"}
```

Die folgende LINQ-Abfrage selektiert die Monatsnamen mit einer Länge von 6 Buchstaben, wandelt sie in Großbuchstaben um und ordnet sie alphabetisch.

```
Dim expr = From s In monate
           Where s.Length = 6
           Order By s
           Select s.ToUpper()
```

Die Ergebnisanzeige:

```
For Each item As String In expr
    ListBox1.Items.Add(item)
Next item
```

Das Resultat in der *ListBox*:

```
AUGUST
JANUAR
```

Obiges Beispiel verdeutlicht das allgemeine Format einer LINQ-Abfrage:

```
From ... < Where ... Order By ... > Select ...
```

Eine LINQ-Abfrage muss immer mit *From* beginnen. Im Wesentlichen durchläuft *From* eine Liste von Daten. Dazu wird eine Variable benötigt, die jedem einzelnen Datenelement in der Quelle entspricht.

HINWEIS Wer die Sprache SQL kennt, der wird zunächst darüber irritiert sein, warum eine LINQ-Abfrage mit *From* und nicht mit *Select* beginnt. Der Grund hierfür ist der, dass nur so ein effektives Arbeiten mit der IntelliSense von Visual Studio möglich ist. Da zuerst die Datenquelle ausgewählt wird, kann die IntelliSense geeignete Typmitglieder für die Objekte der Auflistung anbieten.

Weiterhin erkennen Sie, wie vom neuen Sprachfeature der lokalen Typinferenz (implizite Variablendeklaration) Gebrauch gemacht wird, denn die Anweisung

```
Dim expr = From s In monate ...
```

ist für den Compiler identisch mit

```
Dim expr As IEnumerable(Of String) = From s In monate ...
```

Zwei alternative Schreibweisen von LINQ Abfragen

Grundsätzlich sind für LINQ Abfragen zwei gleichberechtigte Schreibweisen möglich:

- Query Expression-Syntax (Abfrage-Syntax)
- Extension Method-Syntax[1] (Erweiterungsmethoden-Syntax)

Bis jetzt haben wir aber nur die Query Expression-Syntax verwendet. Um die volle Leistungsfähigkeit von LINQ auszuschöpfen, sollten Sie aber beide Syntaxformen verstehen.

BEISPIEL

Die LINQ-Abfrage des obigen Beispiels in *Extension Method-Syntax*.

```
Dim expr =
           monate.Where(Function(s) s.Length = 6).
           OrderBy(Function(s) s).
           Select(Function(s) s.ToUpper())
```

Oder kompakt in einer Zeile:

```
Dim expr = monate.Where(Function(s) s.Length = 6).OrderBy(Function(s) s).
                       Select(Function(s) s.ToUpper())
```

Wie Sie sehen, verwenden wir bei dieser Notation Erweiterungsmethoden und Lambda-Ausdrücke. Aber auch eine Kombination von *Query Expression-Syntax* mit *Extension Method-Syntax* ist möglich.

BEISPIEL

Obiges Beispiel in gemischter Syntax:

```
Dim expr = (From s In monate
            Where s.Length = 6
            Select s.ToUpper()).
            OrderBy(Function(s) s)
```

[1] Die *Extension Method Syntax* wird auch als *Dot Notation Syntax* bezeichnet.

Hier wurde ein Abfrageausdruck in runde Klammern eingeschlossen, gefolgt von der Erweiterungsmethode *OrderBy*. Solange wie der Abfrageausdruck ein *IEnumerable* zurückgibt, kann darauf eine ganze Kette von Erweiterungsmethoden folgen.

Die Query Expression-Syntax (Abfragesyntax) ermöglicht das Schreiben von Abfragen in einer SQL-ähnlichen Weise. Wo immer es möglich ist, empfehlen wir, vor allem der besseren Lesbarkeit wegen, die Verwendung dieser Syntax. Letztendlich konvertiert jedoch der Compiler alle Queries in die andere, auf Erweiterungsmethoden basierende, Syntaxform. Dabei wird z.B. die Filterbedingung *Where* einfach in den Aufruf einer Erweiterungsmethode namens *Where* der *Enumerable*-Klasse übersetzt, die im Namespace *System.Linq* definiert ist.

Allerdings unterstützt die Query Expression-Syntax nicht jeden standardmäßigen Abfrageoperator bzw. kann nicht jeden unterstützen den Sie selbst hinzufügen. In einem solchen Fall sollten Sie direkt die Extension Method-Syntax verwenden.

Abfrageausdrücke unterstützen eine Anzahl verschiedener »Klauseln«, z. B. *Where*, *Select*, *Order By*, *Group By* und *Join*. Wie bereits erwähnt, lassen sich diese Klauseln in die gleichwertigen Operator-Aufrufe übersetzen, die wiederum über Erweiterungsmethoden implementiert werden. Die enge Beziehung zwischen den Abfrageklauseln und den Erweiterungsmethoden, welche die Operatoren implementieren, erleichtert ihre Kombination, falls die Abfragesyntax keine direkte Klausel für einen erforderlichen Operator unterstützt.

Übersicht der wichtigsten Abfrageoperatoren

Die Klasse *Enumerable* im Namespace *System.Linq* stellt zahlreiche Abfrageoperatoren für LINQ to Objects bereit und definiert diese als Erweiterungsmethoden für Typen die *IEnumerable(Of T)* implementieren.

HINWEIS Kommen bei der Extension Method-Syntax (Erweiterungsmethoden-Syntax) Abfrageoperatoren bzw. -Methoden zur Anwendung, so sollten wir bei der Query Expression-Syntax (Abfrage-Syntax) präziser von Abfrage-Klauseln bzw. -Statements sprechen.

Die folgende Tabelle zeigt die wichtigsten standardmäßigen Abfrageoperatoren von LINQ.

Bezeichnung der Gruppe	Operator
Beschränkungsoperatoren (Restriction)	*Where*
Projektionsoperatoren (Projection)	*Select, SelectMany*
Sortieroperatoren (Ordering)	*OrderBy, ThenBy*
Gruppierungsoperatoren (Grouping)	*GroupBy*
Quantifizierungsoperatoren (Quantifiers)	*Any, All, Contains*
Aufteilungsoperatoren (Partitioning)	*Take, Skip, TakeWhile, SkipWhile*
Mengenoperatoren (Sets)	*Distinct, Union, Intersect, Except*
Elementoperatoren (Elements)	*First, FirstOrDefault, ElementAt*

Tabelle 2.1 LINQ-Abfrageoperatoren

Bezeichnung der Gruppe	Operator
Aggregatoperatoren (Aggregation)	*Count, Sum, Min, Max, Average*
Konvertierungsoperatoren (Conversion)	*ToArray, ToList, ToDictionary*
Typumwandlungsoperatoren (Casting)	*OfType T*
Zuweisungsoperator	*Let*

Tabelle 2.1 LINQ-Abfrageoperatoren *(Fortsetzung)*

Die folgende Abbildung illustriert an einem Beispiel, wie einige der bereits im Vorgängerabschnitt diskutierten neuen Sprach-Features in LINQ-Konstrukten zur Anwendung kommen und wie die Abfrage-Syntax vom Compiler in die äquivalente Erweiterungsmethoden-Syntax umgesetzt wird.

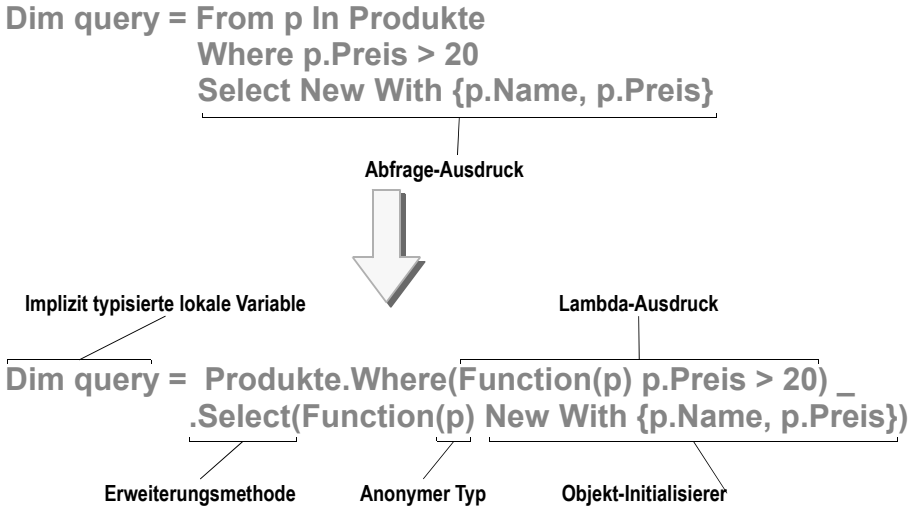

Abbildung 2.4 Vergleich zwischen Abfrage-Syntax (oben) und Erweiterungsmethoden-Syntax (unten)

Beispiele zu LINQ to Objects

Das Ziel der folgenden Beispiele ist nicht die vollständige Erläuterung aller in obiger Tabelle aufgeführten Operatoren und deren Überladungen, sondern vielmehr eine Demonstration des prinzipiellen Aufbaus von Anweisungen zur Abfrage von Objektauflistungen.

In der Regel werden beide Syntaxformen (*Query Expression-Syntax* und Extension *Method-Syntax)* gegenübergestellt, denn nur so erschließt sich am ehesten das allgemeine Verständnis für die auch für den SQL-Kundigen nicht immer leicht durchschaubare Logik der LINQ-Operatoren bzw. -Abfragen.

Für die Beispiele zu LINQ to Objects wird überwiegend auf eine Datenmenge zugegriffen, deren Struktur das folgende, mit dem Klassendesigner von Visual Studio entwickelte, Diagramm zeigt.

HINWEIS Die verwendeten Daten haben ihren Ursprung nicht in einer Datenbank, sondern werden per Code erzeugt (Listing siehe Begleitdateien).

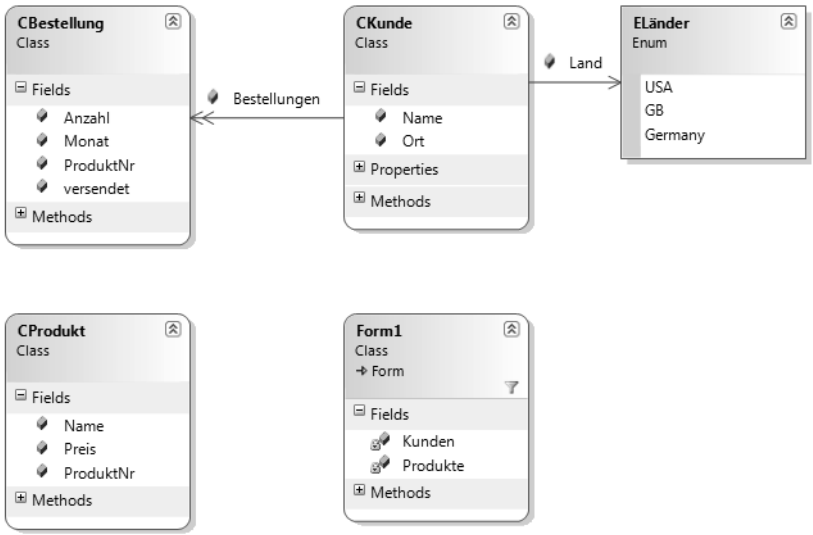

Abbildung 2.5 Das mit dem Klassendesigner entwickelte Klassendiagramm

Die Projektionsoperatoren Select und SelectMany

Diese Operatoren »projizieren« die Inhalte einer Quell-Auflistung in eine Ziel-Auflistung, die das Abfrageergebnis repräsentiert.

Select

Der Operator macht die Abfrageergebnisse über ein Objekt verfügbar, welches *IEnumerable(Of T)* implementiert.

BEISPIEL

Die komplette Produktliste wird ausgegeben. Zunächst in Extension Method-Syntax:

```
Dim allProdukte = Produkte.Select(Function(p) p.Name)
```

Alternativ die Query Expression-Syntax:

```
Dim allProdukte = From p In Produkte Select p.Name
```

Die Ausgabe der Ergebnisliste:

```
For Each p In allProdukte
   ListBox1.Items.Add(p)
Next
```

Der Inhalt der *ListBox* sollte dann etwa folgenden Anblick bieten:

```
Marmelade
Quark
Mohrrüben ...
```

BEISPIEL

Das Abfrageergebnis wird auf einen anonymen Typ projiziert, der als Tupel (Datensatz) definiert ist.

```
Dim expr = Kunden.Select(Function(k) New With {k.Name, k.Ort})
```

Die Ausgabeschleife:

```
For Each k In expr
    ListBox1.Items.Add(k)
Next
```

Das Ergebnis:

```
{Name = Walter, Ort = Altenburg}
{Name = Thomas, Ort = Berlin}
...
```

SelectMany

Stände nur der *Select*-Operator zur Verfügung, so hätte man zum Beispiel bei der Abfrage der Bestellungen für alle Kunden eines bestimmten Landes das Problem, dass das Ergebnis vom Typ *IEnumerable(Of CBestellung)* wäre, wobei es sich bei jedem Element um ein Array mit den Bestellungen eines einzelnen Kunden handeln würde. Um einen praktikableren, d.h. weniger tief geschachtelten, Ergebnistyp zu erhalten, wurde der Operator *SelectMany* eingeführt.

BEISPIEL

Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

```
Dim bestellungen = Kunden.Where(Function(k) k.Land = ELänder.Germany).
                                                    SelectMany(Function(k) k.Bestellungen)
```

Alternativ der Abfrageausdruck in Query Expression Syntax:

```
Dim bestellungen = From k In Kunden
                   Where k.Land = ELänder.Germany
                   From b In k.Bestellungen
                   Select b
```

Das Auslesen des Ergebnisses der Abfrage:

```
For Each b In bestellungen
    ListBox1.Items.Add(b)
Next
```

Das Ergebnis (Voraussetzung ist eine entsprechende Überschreibung der *ToString()*-Methode der Klasse *CBestellung*):

```
ProdNr: 2 , Anzahl: 4 , Monat: März, Versand: False
ProdNr: 1 , Anzahl: 11, Monat: Juni , Versand: True
...
```

Der Restriktionsoperator Where

Dieser Operator schränkt die Ergebnismenge anhand einer Bedingung ein. Sein prinzipieller Einsatz wurde bereits in den Vorgängerbeispielen hinreichend demonstriert. Außerdem können auch Indexparameter verwendet werden, um die Filterung auf bestimmte Indexpositionen zu begrenzen.

BEISPIEL

Die Kunden an den Positionen 2 und 3 der Kundenliste sollen angezeigt werden.

```
Dim expr = Kunden.Where(Function(k, index) (index >= 2) And (index < 4)).Select(Function(k) k.Name)
```

Die Ausgabe:

```
For Each kd In expr
    ListBox1.Items.Add(kd)
Next
```

Das Ergebnis:

```
Holger
Fernando
```

Die Sortierungsoperatoren OrderBy und ThenBy

Diese Operatoren bewirken ein Sortieren der Elemente innerhalb der Ergebnismenge.

OrderBy/OrderByDescending

Das Pärchen ermöglicht Sortieren in auf- bzw. absteigender Reihenfolge.

BEISPIEL

Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (der teuerste zuerst).

```
Dim prod = Produkte
           .Where(Function(p) p.Preis <= 20)
           .OrderByDescending(Function(p) p.Preis)
           .Select(Function(p) New With {p.Name, p.Preis})
```

Oder alternativ als Abfrageausdruck:

```
Dim prod = From p In Produkte
          Where (p.Preis <= 20)
          Order By p.Preis Descending
          Select p.Name, p.Preis
```

Die Ausgabeschleife:

```
For Each p In prod
    ListBox1.Items.Add(p)
Next
```

Das Resultat:

```
{Name = Käse, Preis = 20}
{Name = Mohrrüben, Preis = 15}
...
```

ThenBy/ThenByDescending

Diese Operatoren verwendet man, wenn nacheinander nach mehreren Schlüsseln sortiert werden soll. Da *ThenBy* und *ThenByDescending* nicht auf den Typ *IEnumerable(Of T)*, sondern nur auf den Typ *IOrdered-Sequence(Of T)* anwendbar sind, können diese Operatoren nur im Anschluss an *OrderBy*/*OrderByDescending* eingesetzt werden.

BEISPIEL

Alle Kunden sollen zunächst nach ihrem Land und dann nach ihren Namen sortiert werden.

```
Dim knd = Kunden.OrderBy(Function(k) k.Land).
          ThenBy(Function(k) k.Name).
          Select(Function(k) New With {k.Land, k.Name})
```

Der alternative Abfrageausdruck:

```
Dim knd = From k In Kunden
          Order By k.Land, k.Name
          Select k.Land, k.Name
```

Die Ausgabe

```
For Each ku In knd
    ListBox1.Items.Add(ku)
Next
```

... führt in beiden Fällen zu einem Ergebnis wie diesem:

```
{Land = USA, Name = Fernando}
{Land = USA, Name = Holger}
{Land = GB, Name = Alice}
{Land = Germany, Name = Thomas}
{Land = Germany, Name = Walter}
```

Reverse

Dieser Operator bietet eine einfache Möglichkeit, um die Aufeinanderfolge der Elemente im Abfrageergebnis umzukehren.

BEISPIEL

Das Vorgängerbeispiel mit umgekehrter Reihenfolge der Ergebniselemente:

```
Dim knd = Kunden.OrderBy(Function(k) k.Land).
          ThenBy(Function(k) k.Name).
          Select(Function(k) New With {k.Land, k.Name}).
          Reverse()
```

Ergebnis:

```
{Land = Germany, Name = Walter}
{Land = Germany, Name = Thomas}
...
```

Der Gruppierungsoperator GroupBy

Dieser Operator kommt dann zum Einsatz, wenn das Abfrageergebnis in gruppierter Form zur Verfügung stehen soll. *GroupBy* wählt die gewünschten Schlüssel-Elemente-Zuordnungen aus der abzufragenden Auflistung aus.

BEISPIEL

Alle Kunden nach Ländern gruppieren

```
Dim knd = Kunden.GroupBy(Function(k) k.Land)
```

Der alternative Abfrageausdruck:

```
Dim knd = From k In Kunden Group By k.Land Into Group
```

Durchlaufen der Ergebnismenge:

```
For Each kdGroup In knd
    ListBox1.Items.Add(kdGroup.Land)
    For Each kd In kdGroup.Group
        ListBox1.Items.Add("  " & kd.Name)
    Next
Next
```

Der Gruppenschlüssel (*kdGroup.Key*) ist hier das Land. Die Standardausgabe der Gruppenelemente erfolgt entsprechend der überschriebenen *ToString()*-Methode der Klasse *CKunden* (siehe Beispieldaten zum Buch):

```
Germany
   Walter – Altenburg – Germany
   Thomas – Berlin – Germany
USA
   Holger – Washington – USA
   Fernando – New York – USA
GB
   Alice – London - GB
```

Der *GroupBy*-Operator existiert in mehreren Überladungen, die alle den Typ *IEnumerable(Of IGrouping(Of K, T)* liefern. Die generische Schnittstelle *IGrouping(Of K, T)* definiert einen spezifischen Schlüssel vom Typ *K* für die Gruppenelemente (Typ *T*).

BEISPIEL

Alle Produkte werden nach ihren Anfangsbuchstaben gruppiert.

```
Dim prodGroups = Produkte.GroupBy(Function(p) p.Name(0), Function(p) p.Name)
```

Die (verschachtelte) Ausgabeschleife:

```
For Each pGroup In prodGroups
    ListBox1.Items.Add(pGroup.Key)
    For Each p In pGroup
        ListBox1.Items.Add("  " & p)
    Next
Next
```

Das Ergebnis:

```
M
  Marmelade
  Mohrrüben
  Mehl
Q
  Quark
K
  Käse
H
  Honig
```

Zum gleichen Resultat führt der folgende Code unter Verwendung eines Abfrageausdrucks:

```
Dim prodGroups = From p In Produkte
                 Group By FirstLetter = p.Name(0)
                 Into prods = Group
```

Die (verschachtelte) Ausgabeschleife:

```
For Each pGroup In prodGroups
    ListBox1.Items.Add(pGroup.FirstLetter)
    For Each pr In pGroup.prods
        ListBox1.Items.Add("  " & pr.Name)
    Next
Next
```

Verknüpfen mit Join

Mit diesem Operator definieren Sie Beziehungen zwischen verschiedenen Auflistungen. Im folgenden Beispiel werden Bestelldaten auf Produkte projiziert.

BEISPIEL

Die Bestellungen aller Kunden werden aufgelistet.

```
Dim bestprod = Kunden.
               SelectMany(Function(k) k.Bestellungen).
               Join(Produkte, Function(b) b.ProduktNr,
               Function(p) p.ProduktNr, Function(b, p) New With {b.Monat, p.ProduktNr,
               p.Name, p.Preis, b.versendet})
```

Alternativ die Notation in Abfragesyntax:

```
Dim bestprod = From k In Kunden
               From b In k.Bestellungen
```

```
            Join p In Produkte On b.ProduktNr Equals p.ProduktNr
            Select New With {b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet}
```

Beim Vergleich (*Equals*) ist zu beachten, dass zuerst der Schlüssel der äußeren Auflistung (*b.ProduktNr*) und dann der der inneren Auflistung (*p.ProduktNr*) angegeben werden muss.

Die Anzeigeroutine:

```
For Each bp In bestprod
    ListBox1.Items.Add(bp)
Next
```

Das Ergebnis liefert die Übersicht über alle Bestellungen:

```
{Monat = März, ProduktNr = 2, Name = Quark, Preis = 10, versendet = False}
{Monat = Juni, ProduktNr = 1, Name = Marmelade, Preis = 5, versendet = False}
{Monat = November, ProduktNr = 3, Name = Mohrrüben, Preis = 15, versendet = True}
{Monat = November, ProduktNr = 5, Name = Honig, Preis = 25, versendet = True}
{Monat = Juni, ProduktNr = 6, Name = Mehl, Preis = 30, versendet = False}
{Monat = Februar, ProduktNr = 4, Name = Käse, Preis = 20, versendet = True}
...
```

Aggregatoperatoren

Zum Abschluss unserer Stippvisite bei den LINQ-Operatoren wollen wir noch einen kurzen Blick auf eine weitere wichtige Familie werfen. Diese Operatoren, zu denen *Count*, *Sum*, *Max*, *Min*, *Average* etc. gehören, setzen Sie ein, wenn Sie verschiedenste Berechnungen mit den Elementen der Datenquelle durchführen wollen.

Count

Die von diesem Operator durchzuführende Aufgabe ist sehr einfach, es wird die Anzahl der Elemente in der abzufragenden Auflistung ermittelt.

BEISPIEL

Alle Kunden sollen, zusammen mit der Anzahl der von ihnen aufgegebenen Bestellungen, angezeigt werden.

```
Dim kdn = Kunden.Select(Function(k) New With {k.Name, k.Ort, .AnzahlBest = k.Bestellungen.Count})
```

Oder das Gleiche in Abfrage-Syntax:

```
Dim kdn = From k In Kunden
          Select k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count()
```

Wir iterieren durch die Ergebnismenge:

```
For Each ku In kdn
    ListBox1.Items.Add(ku)
Next
```

Die Ausgabe:

```
{Name = Walter, Ort = Altenburg, AnzahlBest = 1}
{Name = Thomas, Ort = Berlin, AnzahlBest = 2}
...
```

Wie Sie sehen, scheint die Anwendung dieser Operatoren einfach und leicht verständlich zu sein.

Sum

Wie es der Name schon vermuten lässt, können mit diesem Operator verschiedenste Summen aus den Elementen der Quell-Auflistung gebildet werden. Zunächst ein einfaches Beispiel.

BEISPIEL

Die Summe aller Preise der Produktliste

```
Dim total = Produkte.Sum(Function(p) p.Preis)
```

Die alternative Abfrage-Syntax (eigentlich gemischte Syntax):

```
Dim total = (From p In Produkte Select p.Preis).Sum()
```

Die Ausgabe

```
ListBox1.Items.Add(total)
```

... liefert mit den ursprünglichen Beispieldaten den Wert *105*.

Das folgende Beispiel ist nicht mehr ganz so trivial, da sich hier der *Sum*-Operator innerhalb einer verschachtelten Abfrage versteckt.

BEISPIEL

Die Gesamtsumme der von allen Kunden aufgegebenen Bestellungen wird ermittelt.

```
  Dim expr = From k In Kunden
             Join b In
             (
                    From k In Kunden
                    From b In k.Bestellungen
                    Join p In Produkte
                    On b.ProduktNr Equals p.ProduktNr
                    Select New With {k.Name, .BestellBetrag = b.Anzahl * p.Preis}
             ) On k.Name Equals b.Name
             For Each k In expr
                 ListBox1.Items.Add(k.b)
             Next
```

Das Ergebnis:

```
{Name = Walter, BestellBetrag = 40}
{Name = Thomas, BestellBetrag = 340}
...
```

Verzögertes Ausführen von LINQ-Abfragen

Normalerweise werden LINQ-Ausdrücke nicht bereits bei ihrer Definition, sondern erst bei Verwendung der Ergebnismenge ausgeführt. Damit hat man die Möglichkeit, nachträglich Elemente zu der abzufragenden Auflistung hinzuzufügen bzw. zu ändern, ohne dazu die Abfrage nochmals neu erstellen zu müssen.

BEISPIEL

Alle Produkte, die mit dem Buchstaben »M« beginnen, sollen ermittelt werden.

```
Dim MProds = From p In Produkte Where p.Name(0) = "M" Select p.Name
```

Die Ergebnismenge wird das erste Mal durchlaufen und angezeigt:

```
For Each prod In MProds
    ListBox1.Items.Add(prod)
Next prod
ListBox1.Items.Add("----------")
```

Anschließend ändern wir ein Element in der der Abfrage zugrunde liegenden Quelle:

```
Produkte(0).Name = "Milch"
```

... und durchlaufen die Ergebnismenge ein zweites Mal:

```
For Each prod In MProds
    ListBox1.Items.Add(prod)
Next prod
```

Die Ausgabe im Listenfeld zeigt, dass in der zweiten Ergebnismenge das geänderte Element erscheint:

```
Marmelade
Mohrrüben
Mehl
----------
Milch
Mohrrüben
Mehl
```

Wir sehen, dass die definierte Abfrage immer dann neu ausgeführt wird, wenn wir (wie hier in der *For-Each*-Schleife) auf das Abfrageergebnis (*MProds*) zugreifen.

Abfragen dieser Art bezeichnet man deshalb auch als »verzögerte Abfragen«. Mitunter aber ist dieses Verhalten nicht erwünscht, d.h., man möchte das Abfrageergebnis nicht verzögert, sondern sofort nach Definition der Abfrage zur Verfügung haben. Das hätte auch den Vorteil, dass sich die Performance verbessert, weil die Abfrage nicht (wie im Beispiel innerhalb der *For Each*-Schleife) immer wieder zur Ausführung kommt. Abhilfe schafft hier die im nächsten Abschnitt beschriebene Anwendung von Konvertierungsmethoden.

Konvertierungsmethoden

Zu dieser Gruppe gehören *ToArray*, *ToList*, *ToDictionary*, *AsEnumerable*, *Cast* und *ToLookup*. Sowohl die Methoden *ToArray* als auch *ToList* forcieren ein sofortiges Durchführen der Abfrage.

BEISPIEL

Das Vorgängerbeispiel wird wiederholt, diesmal aber wird das Abfrageergebnis in einem Array zwischengespeichert.

```
Dim MProds = (From p In Produkte Where p.Name(0) = "M" Select p.Name).ToArray
...
```

Die Änderung der Quellfolge bleibt jetzt ohne Konsequenz für das Abfrageergebnis:

```
Produkte(0).Name = "Milch"
...
```

Die Ausgabe:

```
Marmelade
Mohrrüben
Mehl
-----------
Marmelade
Mohrrüben
Mehl
```

Der Zuweisungsoperator Let

In einem Abfrageausdruck ist es mitunter nützlich, das Ergebnis einer Zwischenberechnung zu speichern, um dieses dann in untergeordneten Klauseln zu nutzen.

Der *Let*-Operator erzeugt eine neue Bereichsvariable und initialisiert diese mit dem Ergebnis des zugewiesenen Ausdrucks. Wenn die Bereichsvariable ein enumerierbarer Typ ist, kann sie abgefragt werden.

BEISPIEL

In einer Sprüche-Sammlung wird jeder Spruch in ein Array aus Wörtern aufgesplittet. Es werden alle Wörter selektiert, die mit einem Vokal beginnen.

```
Dim sprüche As String() = {"Wer den Cent nicht ehrt ist den Euro nicht wert.",
                           "Müßiggang ist aller Laster Anfang.",
                           "Dummheit und Stolz wachsen auf einem Holz."}

Dim aQuery = From spruch In sprüche
             Let worte = spruch.Split(" "c)
             From wort In worte
             Let w = wort.ToLower()
             Where w(0) = "a" OrElse w(0) = "e" OrElse w(0) = "i"
                              OrElse w(0) = "o" OrElse w(0) = "u"
             Select wort
```

Abfrage ausführen:

```
Dim s As String = Nothing

For Each v In aQuery
        s += v.ToString() & "  "
Next
```

Der Ergebnisstring:

```
ehrt  ist  Euro  ist  aller  Anfang.  und  auf  einem
```

Obiges Beispiel benutzt *Let* auf zweierlei Art:

- Erzeugen eines abfragbaren (enumerierbaren) Typs *worte*
- Die Abfrage braucht *ToLower* nur einmal auf der Bereichsvariablen *wort* aufzurufen. Ohne *Let* müsste *ToLower* für jeden Teil in der *Where*-Klausel aufgerufen werden.

HINWEIS Das komplette Beispiel finden Sie in den Begleitdaten zum Buch!

Abfragen mit PLINQ

Als Reaktion auf die zunehmende Verfügbarkeit von Mehrprozessorplattformen bietet PLINQ eine einfache Möglichkeit, die Vorteile paralleler Hardware zu nutzen. PLINQ ist eine parallele Implementierung von LINQ to Objects und kombiniert die Einfachheit und Lesbarkeit der LINQ Syntax mit der Leistungsfähigkeit der parallelen Programmierung. Es steht das komplette Angebot an Standard-Abfrageoperatoren zur Verfügung, zusätzlich gibt es spezielle Operatoren für parallele Operationen.

HINWEIS In vielen Szenarien kann PLINQ signifikant die Geschwindigkeit von LINQ to Objects-Abfragen steigern, da es alle verfügbaren Prozessoren des Computers effizient nutzt.

Wer bereits mit LINQ vertraut ist, dem wird der Umstieg auf PLINQ kaum Sorgen bereiten. Die Verwendung von PLINQ entspricht meistens exakt der von LINQ-to-Objects und LINQ-to-XML. Sie können beliebige der bereits bekannten Operatoren nutzen, wie zum Beispiel *Join*, *Select*, *Where* usw.

Damit können Sie auch unter PLINQ Ihre bereits vorhandenen LINQ-Abfragen auf gewohnte Weise weiter verwenden, wenn Sie dabei einen wesentlichen Unterschied beachten:

HINWEIS Parallelisieren Sie die Abfrage durch Aufruf der Erweiterungsmethode *AsParallel*!

Die Erweiterungsmethode *AsParallel* gehört zur *System.Linq.ParallelQuery*-Klasse, diese ist in der *System.Core.dll* enthalten und repräsentiert eine parallele Sequenz. *AsParallel* kann auf jeder Datenmenge ausgeführt werden, die *IEnumerable(Of T)* implementiert.

Der Aufruf von *AsParallel* veranlasst den VB-Compiler, die parallele Version der Standard-Abfrageoperatoren zu binden. Damit übernimmt PLINQ die weitere Verarbeitung der Abfrage.

BEISPIEL

Eine einfache LINQ-Abfrage über eine Liste von Integer-Zahlen

```
Dim zahlen = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Oder auch:

```
Dim q = From x in zahlen
```

```
        Where x > 3
        Order By x Descending
        Select x
```

Erst beim Iterieren über die Liste wird die Abfrage ausgeführt:

```
For Each z In q
    ListBox1.Items.Add(z.ToString)             ' 9, 8, 7, 6, 5, 4
Next
```

Um dieselbe Abfrage mittels PLINQ auszuführen, ist lediglich *AsParallel* auf den Daten aufzurufen:

BEISPIEL

Die Abfrage im obigen Beispiel mit PLINQ

```
Dim q = From x in zahlen.AsParallel
        Where x > 3
        Order By x Descending
        Select x
```

Die Abfragen in obigen Beispielen wurden in Query Expression-Syntax geschrieben. Alternativ kann man natürlich auch die Extension Method-Syntax[1] verwenden.

BEISPIEL

Beide obigen Abfragen in Erweiterungsmethoden-Syntax

Einfache LINQ-Version:

```
Dim q = zahlen.
         Where(Function(x) x > 3).
         OrderByDescending(Function(x) x).
         Select(Function(x) x)
```

PLINQ-Version:

```
Dim q = zahlen.AsParallel.
        Where(Function(x) x > 3).
        OrderByDescending(Function(x) x).
        Select(Function(x) x)
```

Nach dem Aufruf der *AsParallel*-Methode führt PLINQ transparent die Erweiterungsmethoden (*Where*, *OrderBy*, *Select* ...) auf allem verfügbaren Prozessoren aus. Genauso wie LINQ realisiert auch PLINQ eine verzögerte Ausführung von Abfragen, d.h., erst beim Durchlaufen der *For Each*-Schleife, beim Direktaufruf von *GetEnumerator*, oder beim Eintragen der Ergebnisse in eine Liste (*ToList*, *ToDictionary*,...) wird die Datenmenge abgefragt. Dann kümmert sich PLINQ darum, dass bestimmte Teile der Abfrage auf verschiedenen Prozessoren laufen, was mit versteckten multiplen Threads umgesetzt wird. Sie als Programmierer brauchen das nicht zu verstehen, Sie merken lediglich an der höheren Performance, dass die Prozessoren besser ausgelastet werden.

[1] Der Compiler konvertiert die Query Expression-Syntax ohnehin in die Extension Method-Syntax, sodass letztendlich bei beiden Syntaxformen Erweiterungsmethoden aufgerufen werden.

Probleme mit der Sortierfolge

Wie sollte es anders sein, bei genauerem Hinsehen werden Sie feststellen, dass es doch nicht ganz so unkompliziert ist, LINQ-Abfragen zu parallelisieren. Ganz abgesehen davon, dass die Parallelisierung nicht immer den erhofften Geschwindigkeitszuwachs bringt, habe wir es noch mit einem schwierigen und vor allem nicht gleich erkennbaren Problem zu tun: der Sortierfolge. Diese bereitet im Zusammenhang mit der parallelen Verarbeitung teilweise recht große Probleme, da auch bei einer geordneten Ausgangsmenge nicht eindeutig ist, in welcher Reihenfolge die Elemente durch PLINQ verarbeitet werden. Je nach LINQ-Operator kann es zu recht merkwürdigen Ergebnissen kommen[1].

Aus diesem Grund wurde die Erweiterungsmethode *AsOrdered* eingeführt. Verwenden Sie diese im Zusammenhang mit *AsParallel,* wird die Sortierfolge der Ausgangsmenge in jedem Fall beibehalten.

BEISPIEL

Verwendung von *AsOrdered*

```
Dim zahlen = {7, 4, 2, 3, 1, 6, 11, 5, 10, 8, 9, 13, 12}
Dim q = (From x In zahlen.AsParallel.AsOrdered
         Where x > 3
         Select x).
         Take(5)                              ' nur die ersten fünf
```

Das Ergebnis wird in jedem Fall

```
7, 4, 6, 11, 5
```

sein. Lassen Sie *AsOrdered* weg, sind weder die obige Reihenfolge noch die Zahlen eindeutig bestimmbar. Unter Umständen könnte auch

```
13, 7, 11, 6, 5
```

ausgegeben werden.

Wie sich Sortierfolgen auf bestimmte Operatoren auswirken, beschreibt im Detail die folgende Webseite:

WWW http://msdn.microsoft.com/de-de/library/dd460677%28VS.100%29.aspx

HINWEIS Grundsätzlich jedoch gilt: Vermeiden Sie im Zusammenhang mit PLINQ die Anwendung von Sortieroperationen, diese machen die Vorteile von PLINQ durch erhöhten Verwaltungsaufwand meist wieder zunichte.

[1] Dies ist auch von der Anzahl der Prozessoren und der Größe der Datenmenge abhängig.

How-to-Beispiele

2.1 ... LINQ-Abfragen verstehen?

LINQ-Abfrage: Query Expression Syntax, Extension Method Syntax; Lambda-Ausdruck; LINQ-Abfrageoperatoren: *Select*, *From*, *Where*, *Order By;* LINQ-Erweiterungsmethoden: *Where*, *OrderBy*, *Select*;

In diesem Lernbeispiel üben Sie den prinzipiellen Aufbau von LINQ-Abfragen. Im Zusammenhang damit kommen die neu eingeführten Sprachfeatures wie Typinferenz, Lambda-Ausdrücke und Erweiterungsmethoden zum Einsatz.

Die zwei grundlegenden Syntaxformen für LINQ-Abfragen:

- *Query* Expression *Syntax*
  Hier werden Standard-Query-Operatoren verwendet
- *Extension Method Syntax*
  Hier kommen Erweiterungsmethoden zum Einsatz

Im Folgenden werden diese beiden Syntaxformen gegenübergestellt, um den Inhalt eines Integer-Arrays zu verarbeiten. Außerdem wird eine Mischform vorgeführt.

Oberfläche

Auf dem Startformular *Form1* finden eine *ListBox* und vier *Button*s ihren Platz (siehe Laufzeitansicht).

Quellcode

```
...
Imports System.Linq
...
Partial Public Class Form1
```

Das abzufragende Integer-Array enthält irgendwelche Werte:

```
    Private zahlen() As Integer = {5, -4, 18, 26, 0, 19, 16, 2, -1, 0, 9, -5, 8, 15, 19 }
```

Die Abfrage in Query Expression Syntax:

```
    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
```

Im Abfrageergebnis sollen alle Zahlen, die größer als 10 sind, enthalten sein und nach ihrer Größe sortiert werden. Im Abfrageausdruck kommen die SQL-ähnlichen Standard-Abfrageoperatoren (*From*, *Where*, *Order By*, *Select*) und so genannte Typinferenz zum Einsatz:

```
        Dim expr = From z In zahlen
                   Where z > 10
                   Order By z
                   Select z
```

Die Anzeige:

```
        For Each z As Integer In expr
                ListBox1.Items.Add(z.ToString())
        Next z

    End Sub
```

Dieselbe Abfrage in Extension Method Syntax:

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
```

Hier werden im Abfrageausdruck so genannte Erweiterungsmethoden (*Where*, *OrderBy*, *Select*) zusammen mit Lambda-Ausdrücken (... *Function(z) z > 10* ...) benutzt:

```
        Dim expr = zahlen.Where(Function(z) z > 10).OrderBy(Function(z) z).Select(Function(z) z)
```

Die Anzeige:

```
        For Each z As Integer In expr
            ListBox1.Items.Add(z.ToString())
        Next z
    End Sub
```

Die gleiche Abfrage in gemischter Syntax:

```
     Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
```

Hier wird der erste Teil des Abfrageausdrucks in Query Expression Syntax, und der zweite (mit einem Punkt eingeleitete) Teil in Extension Method Syntax geschrieben:

```
       Dim expr = ( From z In zahlen
                    Where z > 10
                    Select z).OrderBy(Function(z) z)
```

Die Anzeige:

```
        For Each z As Integer In expr
            ListBox1.Items.Add(z.ToString())
        Next z

    End Sub
```

ListBox-Inhalt löschen:

```
    Private Sub Button4_Click(sender As Object, e As EventArgs) Handles Button4.Click
        ListBox1.Items.Clear()
    End Sub
 End Class
```

Test

Egal, auf welche der drei Schaltflächen Sie klicken, das Ergebnis wird stets dasselbe sein (Abbildung 2.6).

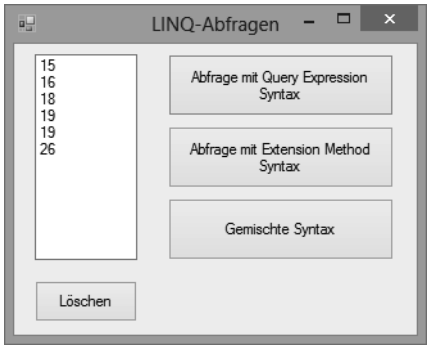

Abbildung 2.6 Laufzeitansicht

2.2 ... nichtgenerische Collections abfragen?

Cast-, *TypeOf*-Operator; LINQ-Abfrage: *ToList*-Methode; *ArrayList*-Objekt;

Wenn Sie das vorhergehende How-to durchgearbeitet haben, sollten Sie in der Lage sein, In-Memory-Collections mit LINQ to Objects abzufragen. Dies gilt allerdings nur für Collections, die das Interface *IEnumerable(Of T)* implementieren, wie beispielsweise *System.Collections.Generic. List(Of T),* sowie Arrays, Dictionaries und Queues. Das Problem besteht darin, dass *IEnumerable(Of T)* ein generisches Interface ist, aber nicht alle Klassen generisch sind.

Ein Beispiel hierfür ist die Datenstruktur *System.Collections.ArrayList.* Hier handelt es sich um eine nichtgenerische Collection, die eine Liste untypisierter Objekte enthält und die *IEnumerable(Of T)* nicht implementiert.

BEISPIEL

Der folgende Versuch, eine *ArrayList* abzufragen, misslingt:

```
Private autoListe As New ArrayList(5)
...
Dim abfrage = From auto In autoListe
              Where auto.Preis > 10000
              Select New With { auto.Typ, auto.Preis, auto.Baujahr }
```

Zur Entwurfszeit erhalten Sie keinerlei Unterstützung durch die Intellisense, was Sie bereits stutzig machen sollte. Beim Kompilieren erscheint eine Fehlermeldung, weil der Typ der Variablen *autoListe* nicht unterstützt wird.

Das vorliegende How-to demonstriert mehrere Lösungsmöglichkeiten.

Oberfläche

Auf das Startformular *Form1* setzen Sie ein *DataGridView*, sowie drei *Button*s (siehe Laufzeitansicht).

ArrayList erzeugen

Bevor wir mit unseren Experimenten beginnen, müssen wir uns zunächst eine geeignete nichtgenerische Collection besorgen.

Wegen unserer *ArrayList* ist folgender Namespace einzubinden:

```
Imports System.Collections
...

   Public Class Form1
```

Wir demonstrieren unser Problem anhand von Objekten einer Klasse *CAuto*, deren Eigenschaften einfachheitshalber als öffentliche Variablen vorliegen:

```
Public Class CAuto
    Public Typ As String
    Public Baujahr As Integer
    Public Preis As Decimal
End Class
```

Eine *ArrayList* mit der Startkapazität 5 wird erzeugt:

```
Private autoListe As New ArrayList(5)
```

Im Konstruktor des Formulars füllen wir die *ArrayList* mit fünf Objekten:

```
Public Sub New()
    InitializeComponent()
```

Das Erzeugen der einzelnen *CAuto*-Objekte wird hier mittels *Objektinitialisierer* realisiert. Diese Neuerung, die ohne einen extra Konstruktor auskommt, ist eine der sprachlichen Voraussetzungen für die LINQ-Technologie:

```
        With autoListe
            .Add(New CAuto With {.Typ = "Ford", .Baujahr = 2005, .Preis = 12000D})
            .Add(New CAuto With {.Typ = "Opel", .Baujahr = 2007, .Preis = 17500D})
            .Add(New CAuto With {.Typ = "Mazda", .Baujahr = 2006, .Preis = 9600D})
            .Add(New CAuto With {.Typ = "Opel", .Baujahr = 2005, .Preis = 7200D})
            .Add(New CAuto With {.Typ = "Ford", .Baujahr = 2008, .Preis = 21700D})
        End With
    End Sub
```

Variante 1 (mit Cast-Operator)

Cast nimmt einen nichtgenerischen *IEnumerable* und liefert Ihnen einen generischen *IEnumerable(Of T)* zurück. Wenn das zurückgegebene Objekt enumeriert ist, iteriert es durch die Quellensequenz und liefert jedes Element als Typ *T*.

```
Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click
    Dim Abfrage = From auto In autoListe.Cast(Of CAuto)()
                  Where auto.Preis > 10000
                  Select New With {auto.Typ, auto.Preis, auto.Baujahr}
    DataGridView1.DataSource = Abfrage.ToList()
End Sub
```

Variante 2 (mit Typisierung)

Unsere Abfrage kann auch ohne *Cast* formuliert werden, indem wir die Iterationsvariable explizit als Typ *CAuto* deklarieren.

```
    Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
        Dim Abfrage = From auto As CAuto In autoListe
                      Where auto.Preis > 10000
                      Select New With {auto.Typ, auto.Preis, auto.Baujahr}
        DataGridView1.DataSource = Abfrage.ToList()
    End Sub
```

Variante 3 (mit OfType)

Als Alternative zum *Cast*-Operator können Sie auch den *OfType*-Operator einsetzen. Der Unterschied besteht darin, dass *OfType* nur die Objekte eines bestimmten Typs aus der Quell-Collection zurückgibt. Hätten Sie zum Beispiel eine *ArrayList* mit *CAuto*- und *CFahrrad*-Objekten, so würde der Aufruf von *ArrayList.OfType(Of CAuto)* nur die Instanzen von *CAuto* aus der *ArrayList* liefern.

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        Dim Abfrage = From auto In autoListe.OfType(Of CAuto)()
                      Where auto.Preis > 10000
                      Select New With {auto.Typ, auto.Preis, auto.Baujahr}
        DataGridView1.DataSource = Abfrage.ToList()
    End Sub
End Class
```

Test

Die drei Varianten zeigen erwartungsgemäß das gleiche Ergebnis (alle Autos mit einem Preis ab 10.000 Euro).

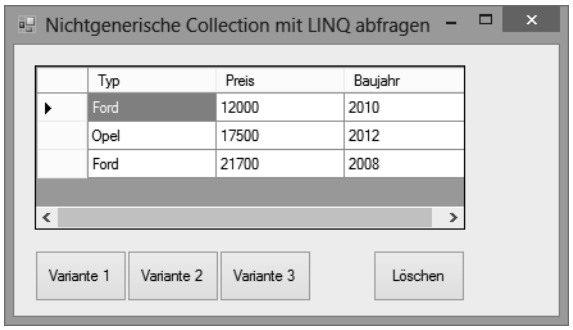

Abbildung 2.7 Laufzeitansicht

HINWEIS In Zukunft werden Sie wahrscheinlich mehr und mehr auf die *ArrayList* und andere nichtgenerische Collections verzichten, denn generische Listen bieten neben einer Typüberprüfung auch eine verbesserte Performance.

2.3 ... Datenbankabfragen mit LINQ und ADO.NET vergleichen?

SQL-Befehle: SELECT, FROM, WHERE; LINQ-Abfrageoperatoren: *Select*, *From*, *Where*

Da in diesem Buch das Haupteinsatzgebiet von LINQ selbstverständlich die Abfrage von Datenbanken ist, wollen wir in diesem abschließendem How-to an einem simplen Beispiel den klassischen Datenbankzugriff mittels ADO.NET mit dem alternativen Zugriff über LINQ to SQL vergleichen.

Ziel ist die Anzeige der Firma (*CompanyName*) und des Landes (*Country*) aller Kunden der *Northwind*-Datenbank, die aus London sind. Der Fokus liegt auf dem Vergleich der Syntax einer klassischen SQL-Abfrage mit der LINQ-Syntax.

Oberfläche

Öffnen Sie eine neue Windows Forms-Anwendung und bestücken Sie das Startformular *Form1* mit einer *ListBox* und drei *Button*s (siehe Laufzeitansicht am Schluss).

Ziehen Sie die Datenbankdatei *Northwind.mdf* per Drag & Drop in den Projektmappen-Explorer. Den sich dabei ungefragt öffnenden Assistenten brechen Sie einfach ab.

Klassischer Datenbankzugriff

Binden Sie zunächst den Namespace *System.Data.SqlClient* ein:

```
Imports System.Data.SqlClient

Public Class Form1

    Private Sub Button1_Click(sender As Object, e As EventArgs) Handles Button1.Click

        Dim connStr As String =
            "Data Source=(LocalDB)\v11.0;AttachDbFilename=|DataDirectory|\Northwind.mdf;" +
                                                    "Integrated Security=True;User Instance=False"
        Dim connection As New SqlConnection(connStr)
        Dim command As SqlCommand = connection.CreateCommand()
```

Die klassische SQL-Abfrage (mit Parameter):

```
        command.CommandText = "SELECT CompanyName, Country FROM Customers  WHERE City = @City"
        command.Parameters.AddWithValue("@City", "London")
```

Öffnen der Datenbankverbindung, Auslesen und Anzeige der Ergebnisse der Abfrage:

```
        connection.Open()

        Dim reader As SqlDataReader = command.ExecuteReader()
        While reader.Read()
            Dim name As String = reader.GetString(0)
            Dim land As String = reader.GetString(1)
            ListBox1.Items.Add(name & "  " & land)
        End While
        reader.Close()
        connection.Close()
    End Sub
```

Datenbankzugriff mit LINQ to SQL

Sie müssen zunächst über das Menü *Projekt/Neues Element hinzufügen...* eine neue *LINQ to SQL-Klasse* hinzufügen. Belassen Sie den standardmäßig vergebenen Namen *DataClasses1.dbml*. Öffnen Sie den Server-Explorer[1] (*Ansicht/Server-Explorer*) und ziehen Sie per Drag & Drop die *Customers*-Tabelle auf die Entwurfsoberfläche des LINQ to SQL-Designers.

```
  Private Sub Button2_Click(sender As Object, e As EventArgs) Handles Button2.Click
      Dim DB As New DataClasses1DataContext
```

Datenbankabfrage als LINQ-Ausdruck:

```
        Dim customers = From cust In DB.Customers
                        Where cust.City = "London"
                        Select cust.CompanyName, cust.Country
```

Ausführen der LINQ-Abfrage und Anzeige der Ergebnisse:

```
        For Each cust In customers
            ListBox1.Items.Add(cust.CompanyName & " " & cust.Country)
        Next
    End Sub
```

Anzeige löschen:

```
    Private Sub Button3_Click(sender As Object, e As EventArgs) Handles Button3.Click
        ListBox1.Items.Clear()
    End Sub
End Class
```

Test

Beide Varianten zeigen erwartungsgemäß das gleiche Ergebnis:

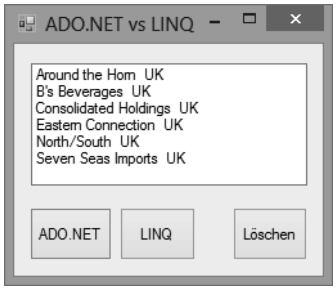

Abbildung 2.8 Laufzeitansicht

Bemerkungen

- Vergleichen Sie die Syntax beider Abfrageformen, so sehen Sie, dass es sich bei LINQ um eine SQL-ähnliche Syntax handelt.
- In diesem Beispiel geht es uns lediglich um das Vermitteln eines kleinen Vorgeschmacks, denn eine systematische Einführung in ADO.NET, SQL und LINQ to SQL erfolgt erst in den Kapiteln 3, 17 und 18.

[1] Bitte nicht mit *SQL Server-Objekt-Explorer* verwechseln!