

HANSER

Leseprobe

Thorsten Kansy

Datenbankprogrammierung mit .NET 3.5

Mehrschichtige Applikationen mit Visual Studio 2008 und MS SQL  
Server 2008

Herausgegeben von Holger Schwichtenberg

ISBN: 978-3-446-41450-1

Weitere Informationen oder Bestellungen unter

<http://www.hanser.de/978-3-446-41450-1>

sowie im Buchhandel.

## 8 Language Integrated Query

Language Integrated Query (LINQ) ist einer der neuen, wesentlichen Bestandteile ab .NET Framework 3.5, mit denen der Entwickler unterschiedlichste Daten mittels einer einheitlichen Abfragesprache verarbeiten kann. LINQ ist dabei in den Sprachen C# und VB.Net direkt eingebaut, sodass der Compiler ein hohes Maß an Kontrolle darüber hat. Die Vorteile sollen an dieser Stelle kurz vorgestellt werden.

- LINQ vereinheitlicht Abfragen auf unterschiedlichste Datenquellen. Ob XML, Data-Sets, Auflistungen, (Sequenzen) oder relationale Datenbanken, die Syntax für den Zugriff ist immer dieselbe.
- T-SQL-Anweisungen werden als Zeichenketten in der Anwendung vorgehalten und als solche an den SQL Server geschickt, ohne dass der Compiler Syntax und Inhalt überwachen kann. Kommt es zu einem Fehler, so geschieht dies erst zur Laufzeit. Bei LINQ ist der Compiler in der Lage, die Syntax zu überwachen.
- LINQ ist typsicher und vollzieht damit den Brückenschlag zwischen Daten und Anwendung. Der Compiler kann zur Entwurfszeit überprüfen, ob Spalten und Rückgabewerte korrekt sind.
- Entwickler müssen sich nur noch in eine Sprache wie C# einarbeiten. Es ist nicht absolut zwingend notwendig, über SQL-Kenntnisse zu verfügen (jedoch bestimmt von großem Vorteil).

LINQ selbst ist, wie schon beschrieben, Bestandteil der Sprachen C# und VB.Net und kann dabei für extrem unterschiedliche Datengrundlagen zum Einsatz kommen. Praktisch bedeutet dies, dass eine Abfrage wie die folgende (korrekt) ausgeführt werden kann, unabhängig, ob diese auf XML-Daten oder Daten aus einer SQL Server-Datenbank zugreift.

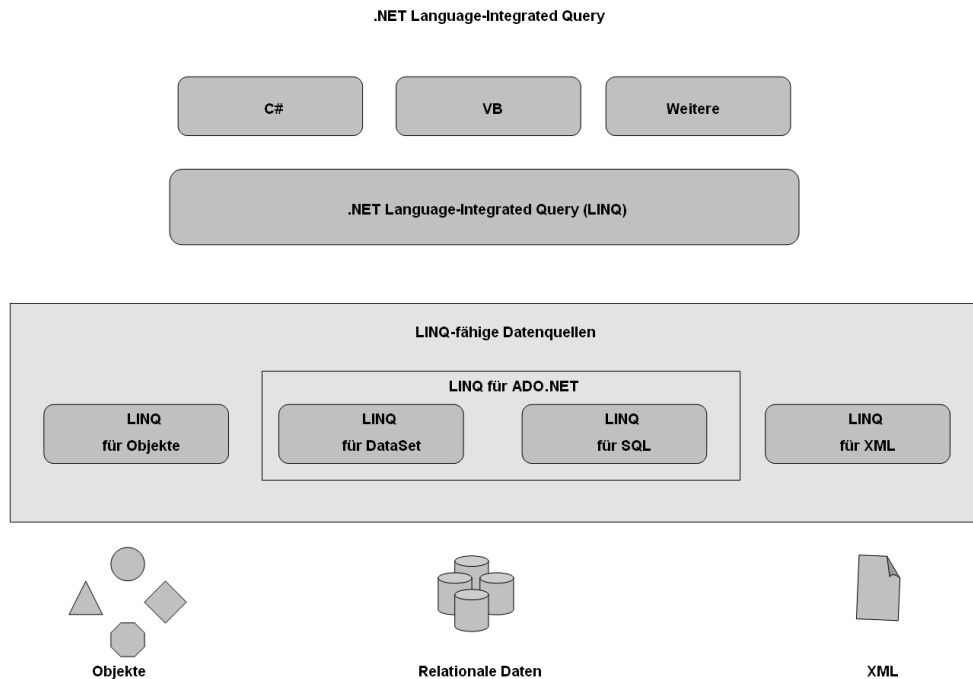
```
var autos =  
    from a in Autos  
    where a.Marke == "Mazda"  
    select new { a.Marke, a.Baujahr}
```

Wichtig ist nur, dass die Struktur der Daten identisch ist, da sonst der Compiler einen Fehler meldet.

**Hinweis:**

Erklärungen zu den Neuheiten in C# 3.x, von denen LINQ zum Teil regen Gebrauch macht, finden sich unter 2.3, „Neuheiten in C# 3.x“.

Damit dies möglich ist, bedient sich LINQ eines Mehrschichtenmodells, in dem die Abfragesprache nur die oberste Schicht darstellt. Über eine Vielzahl von datenquellenspezifischen Implementierungen werden die Daten aus sogenannten LINQ-fähigen Datenquellen (LINQ enabled Data Sources) bereitgestellt. Das folgende Schema zeigt den Aufbau.



**Abbildung 8.1** LINQ im Überblick

Durch diese Trennung zwischen Zugriffssprache und Datenquellen wird es in Zukunft möglich sein, weitere Quellen zu implementieren. (Z.B. ist für ca. 10/08 „LINQ to Entities“ von Microsoft angekündigt.)

Folgende LINQ-fähige Datenquellen gibt es zurzeit:

- *LINQ to SQL*: LINQ-Abfragen werden zu SQL-Anweisungen umgewandelt und gegen eine relationale Datenbank ausgeführt. Daten können so abgefragt und auch dauerhaft geändert werden. Diese Variante wird grob unter „LINQ to ADO.NET“ eingeordnet.
- *LINQ to DataSets*: Über LINQ können hier Daten verarbeitet werden, die bereits im Speicher in Form von DataSets liegen.

- *LINQ to Objects*: LINQ-Abfragen können auf allen Auflistungen durchgeführt werden, welche die `System.Collection.Generic.IEnumerable<T>`-Schnittstelle implementieren.
- *LINQ to XML*: LINQ-Abfragen können auf XML-Dokumente angewendet werden.

Der Code in LINQ-Syntax wird intern vor der Kompilierung in eine Abfolge der entsprechenden Methoden umgewandelt. So wird z.B. die folgende Abfrage

```
var expr =
    from k in Kunden
    where (k.Sprache == eSprache.Deutsch)
    orderby (k.Name)
    select (k);
```

zu diesem Code umgesetzt:

```
var expr = Kunden
    .Where (k=> k.Sprache == eSprache.Deutsch)
    .OrderBy (k=> k.Name)
    .Select (k);
```

Eine Reihe von Funktionalitäten (z.B. das Sortieren nach mehr als nur einem Wert) ist sogar nicht direkt über die LINQ-Syntax verfügbar, sondern nur direkt über die dahinter stehenden Methoden. In diesen Fällen wird in den folgenden Abschnitten auf die punktorientierte Syntax zurückgegriffen, wie sie sonst auch bei „gewöhnlichen“ Methoden verwendet wird.

#### Hinweis:

Der Grund, warum eine LINQ-Abfrage im Gegensatz zu einer SQL-Abfrage mit `from` beginnt, ist der, dass auf diese Weise das Visual Studio 2008 mit IntelliSense den Entwickler mit Vorschlägen unterstützen kann. Damit wird das Schreiben von Bedingungen, Auswahllisten etc. beschleunigt.

Um das Ergebnis schließlich auszugeben, können `foreach`-Schleifen verwendet werden. Hier eine einfache Schleife. Es gibt jedoch auch Situationen, in denen mehrere Schleifen ineinander verschachtelt werden müssen.

```
foreach (var i in qErgebnis)
    Debug.WriteLine(i);
```

In diesem Kapitel werden lediglich die Fälle gezeigt, bei denen die Informationen mit dem `Debug`-Objekt ausgegeben werden, um den Effekt der jeweiligen Operatoren zu demonstrieren. Da die Ergebnissequenz jedoch ebenfalls die `System.Collection.Generic.IEnumerable<T>`-Schnittstelle implementiert, stehen neben dem Durchlaufen in einer `foreach`-Schleife auch alle anderen Methoden offen, die diese Schnittstelle voraussetzen.

#### Hinweis:

Wenn Sie die Informationen, die angezeigt werden, nicht für sehr sprechend halten (z.B. weil für jedes Objekt nur der Typ ausgegeben wird), dann bedenken Sie, dass die `ToString()`-Methode überschrieben werden kann, um dieses Manko auszuwetzen. Für die Klasse `cKunde` kann dies z.B. so aussehen:

```
public override string ToString()
{
    return string.Format("{0}: {1} aus {2}", ID, Name, Ort);
}
```

Im Code zu diesem Kapitel wurde dies für die Klassen `cKunde`, `cWare` und `cBestellung` getan, ohne dass dies hier abgedruckt wurde, damit der Blick auf das Wesentliche erhalten bleibt.

Werden anonyme Typen ausgegeben, so wird die `ToString()`-Methode automatisch so überschrieben, dass diese einfach alle Eigenschaften per Name und aktuellem Wert aufführt.

## 8.1 LINQ-Abfragen

In diesem Abschnitt wird alles Wichtige über Abfragen mittels LINQ beschrieben. Da diese Daten am einfachsten zu handhaben sind, werden zum Speichern der Daten Klassen verwendet, welche die Schnittstelle `System.Collection.Generic.IEnumerable<T>` implementieren. Da die Sprachelemente immer die gleichen sind (einer der Vorteile von LINQ), werden die notwendigen Schritte und feinen Unterschiede für den Zugriff auf unterschiedliche Datenquellen in den Abschnitten 8.2, „LINQ to SQL“ und 8.3, „LINQ to DataSets“, beschrieben. Weitere LINQ-Versionen wie z.B. LINQ to XML und LINQ für Entities werden in diesem Buch nicht näher beschrieben.

### 8.1.1 Klassen/Tabellen für die Beispiele

Für die gesamten Beispiele über LINQ-Abfragen kommt diese einfache Datenstruktur auf Basis von Klassen zum Zuge. Die folgende Abbildung zeigt die Zusammenhänge der vier Klassen, die für LINQ to SQL im weiteren Verlauf des Kapitels auch als Tabellen abgebildet werden. Im Folgenden sind die Beziehungen als Tabellendiagramm abgebildet.

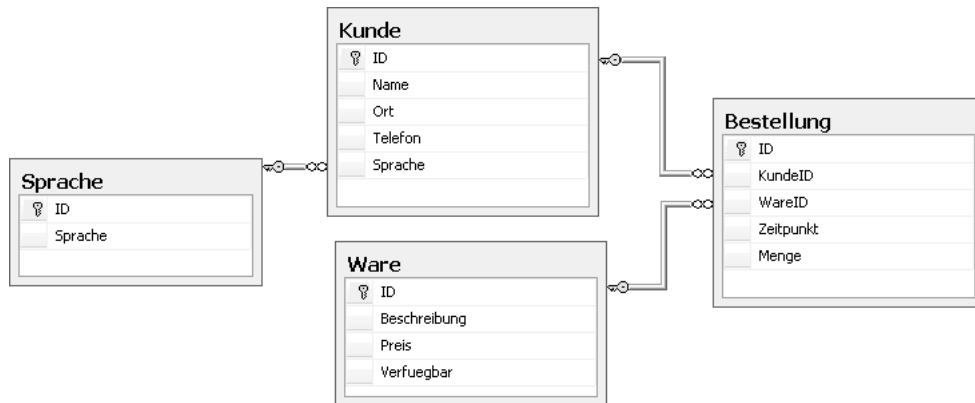


Abbildung 8.2 Die Zusammenhänge der vier Klassen/Tabellen

Entsprechend werden die Klassen im folgenden Quelltext definiert:

```
// Aufzählung der Sprachen der Kunden
public enum eSprache
{
    Daenisch = 1030,
    Deutsch = 1031,
    EnglischUSA = 1033,
    Japanisch = 1041
}

// Ein Kunde
public class cKunde
{
    public int ID;
    public string Name;
    public string Ort;
    public string Telefon;
    public eSprache Sprache;
}

// Eine Ware
public class cWare
{
    public int ID;
    public string Beschreibung;
    public decimal Preis;
    public bool Verfuegbar;
}

// Eine Bestellung eines Kunden
public class cBestellung
{
    public int ID;
    public int KundeID;
    public int WareID;
    public DateTime Zeitpunkt;
    public int Menge;
}
```

Nun noch die Initialisierung, damit ein paar Daten für die Abfragen zur Verfügung stehen:

```
// Auflistungen initialisieren
cKunde[] Kunden = new cKunde[]
{
    new cKunde { ID=1, Name="Müller AG", Ort="Bochum",
        Telefon="+49 234-123456", Sprache=eSprache.Deutsch },
    new cKunde { ID=2, Name="Yokai Inc", Ort="Tokio",
        Telefon="+81 103 123456", Sprache=eSprache.Japanisch },
    new cKunde { ID=3, Name="Smith Inc", Ort="Detroit",
        Telefon="+1 234-123456", Sprache=eSprache.EnglishUSA },
    new cKunde { ID=4, Name="Meyer AG", Ort="München",
        Telefon="+49 89-123456", Sprache=eSprache.Deutsch },
    new cKunde { ID=5, Name="Hanse AG", Ort="Hamburg",
        Telefon="+49 40-123456", Sprache=eSprache.Deutsch }
};

cWare[] Waren = new cWare[]
{
    new cWare { ID=100, Beschreibung="Wasserbett", Preis=499.90M,
        Verfuegbar=true },
    new cWare { ID=101, Beschreibung="Blumentopf", Preis=9.90M,
        Verfuegbar=false },
    new cWare { ID=102, Beschreibung="Tisch", Preis=49.90M,
        Verfuegbar=true },
    new cWare { ID=103, Beschreibung="Stuhl", Preis=39.90M,
        Verfuegbar=true },
    new cWare { ID=104, Beschreibung="Teppich (groß)", Preis=59.90M,
        Verfuegbar=true }
};
```

```

        Verfuegbar=true },
    new cWare { ID=105, Beschreibung="Teppich (klein)", Preis=49.90M,
        Verfuegbar=true }
};

cBestellung[] Bestellungen = new cBestellung[]
{
    new cBestellung { ID=1000, KundeID=1, WareID=100, Menge=1,
        Zeitpunkt=new DateTime(2007,11,13) },
    new cBestellung { ID=1001, KundeID=2, WareID=102, Menge=1,
        Zeitpunkt=new DateTime(2007,11,14) },
    new cBestellung { ID=1002, KundeID=2, WareID=103, Menge=4,
        Zeitpunkt=new DateTime(2007,11,13) },
    new cBestellung { ID=1003, KundeID=4, WareID=105, Menge=2,
        Zeitpunkt=new DateTime(2007,10,5) },
    new cBestellung { ID=1004, KundeID=3, WareID=104, Menge=1,
        Zeitpunkt=new DateTime(2007,11,7) },
    new cBestellung { ID=1005, KundeID=1, WareID=100, Menge=1,
        Zeitpunkt=new DateTime(2007,11,16) }
};

```

**Hinweis:**  
 Sie finden die Quelltexte für die Deklaration und Initialisierung der Auflistungen im Projekt LINQ zu diesem Kapitel.

Mit dieser Struktur und diesen Daten werden nun im Folgenden die Operatoren (LINQ-Befehle) erklärt; später dann mit einer ähnlichen Tabellenstruktur LINQ to SQL.

### 8.1.2 LINQ-Operatoren

Befehle für LINQ werden Operatoren genannt. Viele dieser Operatoren sind als Erweiterungsmethoden für die generische Schnittstelle `IEnumerable<T>` implementiert.

Im Folgenden findet sich eine kurze Übersicht über die Operatoren, die in LINQ Anwendung finden. Im Anschluss daran folgen vertiefende Erklärungen mit Beispielen.

**Tabelle 8.1** LINQ-Operatoren

Operator	Beschreibung	SQL Äquivalent
Aggregate	Erlaubt die Erstellung benutzerdefinierter Aggregatfunktionen, mit denen Sequenzen durchlaufen werden.	Benutzerdefinierte Aggregatfunktion
All	Liefert <code>true</code> , wenn alle Elemente einer Sequenz die angegebene Bedingung erfüllen.	–
Any	Liefert <code>true</code> , wenn mindestens ein Element der Sequenz die angegebene Bedingung erfüllt.	EXISTS-Funktion
Average	Bildet den Mittelwert (arithmetischen Durchschnitt) einer Sequenz.	AVG-Funktion
Cast	Versucht, alle Elemente einer Sequenz in einen anderen Typen umzuwandeln (casten).	–

Operator	Beschreibung	SQL Äquivalent
Concat	Verkettet zwei Sequenzen zu einer einzigen.	UNION-Schlüsselwort
Contains	Prüft, ob eine Sequenz ein bestimmtes Element enthält.	IN-Schlüsselwort
Count	Ermittelt die Anzahl der Elemente in einer Sequenz. Der Wert ist vom Typ <code>int</code> .	COUNT-Funktion
LongCount	Ermittelt die Anzahl der Elemente in einer Sequenz. Der Wert ist vom Typ <code>long</code> .	COUNT-Funktion
Distinct	Liefert aus einer Sequenz die eindeutigen Elemente. Dubletten werden entfernt	DISTINCT-Schlüsselwort
ElementAt	Liefert das Element in einer Sequenz an einer bestimmten Stelle (Index).	–
ElementAtOrDefault	Liefert das Element in einer Sequenz an einer bestimmten Stelle (Index) oder einen Standardwert, wenn der Index negativ oder größer als die Anzahl der Elemente ist.	–
Empty	Erstellt eine leere Sequenz vom angegebenen Typus.	–
Except	Vergleicht zwei Sequenzen und liefert nur diejenigen Elemente, die nur in einer der beiden Sequenzen vorhanden sind.	–
First	Liefert das erste Element einer Sequenz.	–
FirstOrDefault	Liefert das erste Element einer Sequenz oder einen Standardwert, wenn die Sequenz leer ist.	–
GroupBy	Gruppiert eine Sequenz.	GROUP BY-Schlüsselwort
GroupJoin	Verbindet zwei Sequenzen durch einen OUTER JOIN.	JOIN-Schlüsselwort
Intersect	Liefert die Schnittmenge zweier Sequenzen.	–
Join	Verbindet zwei Sequenzen durch einen INNER JOIN.	JOIN-Schlüsselwort
Last	Liefert das letzte Element einer Sequenz.	–