

Kapitel 10

Language Integrated Query (LINQ)

In diesem Kapitel:

Einführung und Motivation	502
LINQ-Provider	503
LINQ to Objects	514
Parallel LINQ (PLINQ)	527
LINQ to XML	529
LINQ to DataSet	529
LINQ to SQL	530
LINQ to Entities	530
LINQ to DataServices	530

Einführung und Motivation

Language Integrated Query (LINQ) ist eine allgemeine Such-/Abfragesprache, die schon seit dem .NET Framework 3.5 in der .NET-Klassenbibliothek und der Sprachsyntax der Sprachen C# (seit Version 3.0) und Visual Basic .NET (seit Version 9.0) verankert ist.

HINWEIS In .NET 4.0 wurde LINQ um Parallel LINQ (PLINQ) erweitert. Dies wird in diesem Kapitel behandelt. Es gab zudem erhebliche Verbesserungen an der LINQ-Variante *LINQ to Entities*, siehe dazu Kapitel 12 »Objektrelationales Mapping (ORM) mit dem ADO.NET Entity Framework 4.0«.

Das Problem, das LINQ zu lösen versucht, lässt sich so beschreiben: Jede Art von Datenspeicher (z.B. Objektmengen im Hauptspeicher, Datenbanktabellen, XML-Dokumente, Verzeichnisdienste) besitzt eine Möglichkeit zur Suche nach Elementen. Bei Datenbanken ist dies in der Regel die Sprache Structured Query Language (SQL), bei XML-Dokumenten XPath oder XQuery und bei Verzeichnisdiensten LDAP. Für Objektmengen im Hauptspeicher gibt es keinen Standard oder De-Facto-Standard. Innerhalb des .NET Framework findet man unterschiedliche Such- und Abfragemöglichkeiten, z.B. DataView-Objekte für DataTable-Objekte. Auch die Methoden Find() und FindAll(), mit denen man unter Angabe eines Prädikats (vgl. Kapitel 6 »Sprachsyntax Visual Basic 2010 (VB.NET 10.0) und C# 2010 (C# 40)«) in Objektmengen aus dem Namensraum System.Collections suchen kann, lassen sich dabei als eine Abfragesprache bezeichnen. Alle diese Abfragesprachen unterscheiden sich hinsichtlich ihrer Mächtigkeit und auch hinsichtlich ihrer Syntax, sodass man für diese verschiedenen Datenspeicher unterschiedliche Befehlssätze beherrschen muss. erinnert sei an dieser Stelle auch noch daran, dass es zwar einen Standard für SQL gibt, aber es dennoch Unterschiede zwischen der SQL-Syntax verschiedener Datenbankmanagementsysteme gibt.

LINQ tritt an, eine allgemeine Such- und Abfragesyntax für alle Arten von Datenspeichern zu definieren. Unterhalb der LINQ-Abfrageebene werden die Abfragen durch LINQ-Provider in andere Sprachen (z.B. SQL, XPath oder LDAP) übersetzt oder direkt auf dem Datenspeicher ausgeführt.

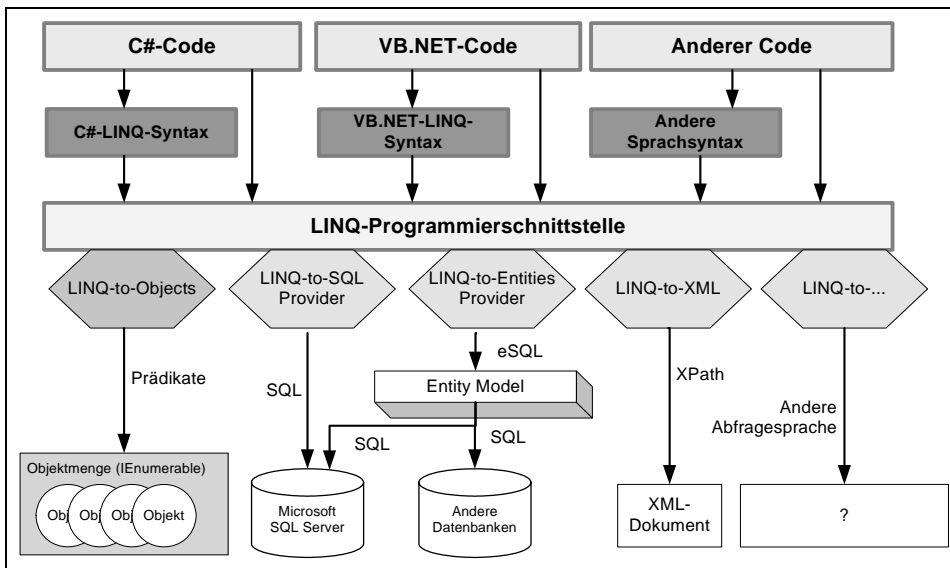


Abbildung 10.1 Architektur von LINQ

Neben der Vereinheitlichung der Sprachen bietet LINQ noch einen Vorteil: Während bisher Sprachen wie SQL, XPath und LDAP aus der Sicht des Sprachcompilers nur Zeichenkettenliterals waren, ist die Abfragesyntax nun in der Sprachsyntax bzw. Klassenbibliothek verankert. Der große Vorteil von LINQ ist, dass die Sprachcompiler die Syntax prüfen können und die Entwicklungsumgebung IntelliSense-Unterstützung anbieten kann. Dies ist mit »externen« Suchsprachen, die der Compiler nur als Zeichenkette sieht, nicht möglich.

LINQ-Provider

Dieser Abschnitt dokumentiert die zum Redaktionsschluss verfügbaren und dem Autor bekannten LINQ-Provider.

HINWEIS LINQ-Provider haben meistens einen Namen, der mit *LINQ to ...* beginnt (z.B. LINQ to XML). Einige wenige Provider verwenden noch die alte Benennungsweise mit einem vorangestellten Kürzel (z.B. hieß LINQ to XML früher *XLINQ*).

LINQ-Provider von Microsoft im .NET Framework

Microsoft bietet seit .NET 3.5 die Möglichkeit zur Abfrage von

- .NET-Objektmengen, die die Schnittstelle `IEnumerable` unterstützen (LINQ to Objects)
- Microsoft SQL Server-Datenbanken (LINQ to SQL, früher: *DLINQ*),
- ADO.NET-Datasets (LINQ to DataSet) und
- XML-Daten (LINQ to XML, früher: *XLINQ*).

Seit .NET 3.5 Service Pack 1 liefert Microsoft noch zusätzlich:

- LINQ to Entities: Abfrage von relationalen Datenbanken (nicht nur Microsoft SQL Server wie bei LINQ to SQL)
- LINQ to DataService: Steuerung von WCF-Datendiensten

Die beiden letztgenannten Provider wurden in .NET 4.0 erweitert.

Außerdem ist neu in .NET 4.0 *Parallel LINQ (PLINQ)* – siehe Abschnitt dazu in diesem Kapitel.

Andere LINQ-Provider

An der breiten Akzeptanz zeigt sich, dass LINQ auf einem guten Weg ist, sich für die Abfrage von Datenquellen unterschiedlichster Art zu etablieren. Mittlerweise gibt es neben den in .NET Framework seit Version 3.5 mitgelieferten Providern eine Reihe von Anbietern (kommerzielle und Open Source), so genannte LINQ-Provider für Ihre Datenquellen.

Die nachfolgende Liste gibt einen guten Überblick über das Spektrum der Möglichkeiten, erhebt jedoch keinen Anspruch auf Vollständigkeit. Hingewiesen sei außerdem darauf, dass viele der hier genannten Provider noch experimentell sind.

Andere LINQ-Provider (1) (Programmier-)Sprachen

- LINQ to XSD
<http://linqtoxsd.codeplex.com/>
- LINQ to Java
<http://xircles.codehaus.org/projects/quaere>
- LINQ to PHP (PHPLINQ)
<http://www.codeplex.com/PHPLinq>
- LINQ to JavaScript (JSLINQ)
<http://www.codeplex.com/JSLINQ>
- LINQ to RDF
<http://blogs.msdn.com/hartmutm/archive/2006/07/24/677200.aspx>
- LINQ to WMI
<http://bloggingabout.net/blogs/emile/archive/2005/12/12/10514.aspx>
- LINQ to LDAP
<http://www.hookedonlinq.com/LINQ2LDAP.ashx>

Andere LINQ-Provider (2) Datenbanken und ORM-Werkzeuge

- LINQ to MySQL, Oracle and PostgreSQL
http://code2code.net/DB_Linq/
- LINQ to Open Access (kommerziell)
<http://www.telerik.com/products/orm.aspx>
- LINQ to Genome (kommerziell)
<http://www.genom-e.com/Default.aspx?tabid=227>
- NHibernate LINQ
<http://ayende.com/Blog/archive/2009/07/26/nhibernate-linq-1.0-released.aspx>
- LINQ to LLBLGen Pro
<http://weblogs.asp.net/fbouma/archive/2008/03/12/beta-of-linq-to-llblgen-pro-released.aspx>

Andere LINQ-Provider (3) Anwendungen und Server

- LINQ to SAP (kommerziell)
<http://www.theobald-software.com>
- LINQ to SharePoint
<http://www.codeplex.com/LINQtoSharePoint>
- LINQ to Microsoft CRM
<http://www.codeplex.com/LinqtoCRM>

- LINQ to Excel
<http://www.codeplex.com/xslinq>
- LINQ to Active Directory
<http://www.codeplex.com/LINQtoAD>

Andere LINQ-Provider (4) Websites

- LINQ to Amazon
<http://weblogs.asp.net/fmarguerie/archive/2006/06/26/Introducing-Linq-to-Amazon.aspx>
- LINQ to Flickr
<http://spellcoder.com/blogs/bashmohandes/archive/2007/04/08/6552.aspx>
- LINQ to Google
<http://glinq.codeplex.com/>
- LINQ to Freebase
<http://metawebtolinq.codeplex.com/>

Formen von LINQ

Es gibt zwei grundsätzliche Formen der LINQ-Unterstützung:

- Abfrage über Mengen, die `IEnumerable` unterstützen: Diese Abfragen fallen alle unter LINQ to Objects und werden von LINQ im RAM ausgeführt.
- Abfrage über Mengen, die `IQueryable` unterstützen: Diese Abfrage werden von einer datenquellenspezifischen LINQ-Implementierung ausgeführt. LINQ übergibt dieser Implementierung die Abfrage in Form eines Ausdrucksbaums (*Expression Tree*). Es ist der Implementierung überlassen, wie die Abfrage erfolgt (z. B. Umsetzung in SQL oder XPath oder Aufruf eines Webservices). Der Einsatz von `IQueryable` ist wesentlich komplexer als der Einsatz von `IEnumerable`, denn bei `IQueryable` werden die LINQ-Abfragen zunächst in einen Ausdrucksbaum (*Expression Tree*) umgewandelt. Dieser sprachneutrale Ausdrucksbaum wird dann an den LINQ-Provider übergeben, der diesen Baum in die jeweilige providerspezifische Anfragesyntax übersetzt.

HINWEIS

Die Erstellung von LINQ-Providern mithilfe von `IQueryable` ist in der MSDN-Dokumentation hinterlegt. Nicht dokumentiert hat Microsoft aber die Erstellung von eigenen LINQ to SQL-Providern, sodass man die Basisimplementierung des LINQ to SQL-Providers auch mit anderen Datenbanken nutzen könnte. Man muss hier also einen LINQ-Provider von Grund auf schreiben, um auf andere Datenbanken zuzugreifen.

Einführung in die LINQ-Syntax

Es gibt zwei Syntaxformen für LINQ: Die Abfragesyntax (Originalbezeichnung: Query Expression Syntax) und die Methodensyntax (Originalbezeichnung: Extension Method Syntax). Die Abfragesyntax ist eleganter, in der Praxis muss man in vielen Fällen beide Syntaxformen mischen, denn viele Befehle sind nur in der Methodensyntax verfügbar.

LINQ-Abfragesyntax

Die Grundstruktur eines LINQ-Befehls in der Abfragesyntax ist

```
from ... where ... orderby ... select ...
```

Die Syntax von LINQ ist an die Datenbankabfragesprache SQL angelehnt, allerdings wird das `from` immer vorangestellt. Der Grund für diese Abweichung von SQL liegt darin, dass Entwicklungsumgebungen in der Lage sein sollen, dem Entwickler Hilfen bei der Eingabe (IntelliSense) zu geben. Dies kann eine Entwicklungsumgebung aber nur, wenn zu Beginn klar ist, auf welche Menge sich die Abfrage bezieht. Dies ist aber nicht die einzige Abweichung von der SQL-Syntax.

Die folgende Beschreibung liefert eine komplette formale Definition der LINQ-Abfragesyntax. Alle diese hier genannten Begriffe (außer den Platzhaltern `id`, `expr`, `source`, `key`, `query`, `condition` und `ordering`) sind Schlüsselwörter der Sprache C# (seit 3.0) bzw. Visual Basic (seit 9.0) und werden von der Entwicklungsumgebung Visual Studio (seit 2008) auch wie Sprachschlüsselwörter eingefärbt.

```
from id in source
{ from id in source |
  join id in source on expr equals expr [ into id ] |
  let id = expr |
  where condition |
  orderby ordering, ordering, ... }
select expr | group expr by key
[ into id query ]
```

Listing 10.1 Syntaxbeschreibung für die LINQ-Abfragesyntax (C#)

```
From id In source
{ from id In source |
  Join id in source On expr Equals expr [ Into id ] |
  Let id = expr |
  Where condition |
  Take x |
  Skip x |
  Order By ordering, ordering, ... }
Select expr | Group expr By key
Aggregate x in source
[ Into id query ]
Distinct
```

Listing 10.2 Syntaxbeschreibung für die LINQ-Abfragesyntax (VB)

An den obigen Syntaxbeschreibungen wird deutlich, dass gar nicht alle Sprachelemente von SQL in der LINQ-Abfragesyntax (d.h. durch eigene Sprachelemente) unterstützt werden. Beispielsweise fehlen in C# `DISTINCT` und `TOP`. Dies bedeutet aber nicht, dass diese Funktionalität in LINQ-Abfragen nicht verfügbar wäre. Es bedeutet nur, dass sie in der LINQ-Abfragesyntax nicht verfügbar sind. Es gibt aber noch eine LINQ-Methodensyntax. In Visual Basic existieren mehr Befehle in der Abfragesyntax.

Beispiele

Vor der Diskussion der Methodensyntax sollen zunächst zwei Beispiele (jeweils in C# und Visual Basic) gezeigt werden.

Beispiel: Abfrage einer Menge von Zeichenketten

In diesem ersten Beispiel werden aus einer Liste von Monaten diejenigen Monate gefiltert, deren Namen vier Zeichen lang sind. Von den Monatsnamen werden nur die ersten drei Zeichen weiterverarbeitet. Die Liste wird lexikalisch aufsteigend sortiert. Das Ergebnis ist also *Jul*, *Jun* und *Mär*.

```
public static void Beispiel1()
{
    // Datendefinition (=Datenquelle)
    string[] AlleMonate = { "Januar", "Februar", "März", "April", "Mai", "Juni", "Juli", "August",
"September", "Oktober", "November", "Dezember" };

    // LINQ-Abfrage
    IEnumerable<string> Monate4 = from Monat in AlleMonate
                                where Monat.Length == 4
                                orderby Monat
                                select Monat.Substring(0, 3);

    // Nutzung des Abfrageergebnisses
    foreach (string Monat in Monate4)
    {
        Console.WriteLine(Monat);
    }
}
```

Listing 10.3 Filtern in einer Liste von Zeichenketten (C#)

```
Public Sub Beispiel1()
    ' Datendefinition (=Datenquelle)
    Dim AlleMonate As String() = { "Januar", "Februar", "März", "April", "Mai", "Juni", "Juli", "August",
                                   "September", "Oktober", "November", "Dezember" }

    ' LINQ-Abfrage
    Dim Monate4 As IEnumerable(Of String) = From Monat In AlleMonate _
        Where Monat.Length = 4 _
        Order By Monat _
        Select Monat.Substring(0, 3)

    ' Nutzung des Abfrageergebnisses
    For Each Monat As String In Monate4
        Console.WriteLine(Monat)
    Next
End Sub
```

Listing 10.4 Filtern in einer Liste von Zeichenketten (VB)

Beispiel: Abfrage einer Menge von Objekten des Typs Process

Im zweiten Beispiel werden aus der Liste der laufenden Prozesse diejenigen herausgefiltert, die weniger als 700.000 Bytes Speicher benötigen. Die Datenmenge wird in diesem Fall von der statischen Methode `GetProcesses()` in der FCL-Klasse `System.Diagnostics.Process` geliefert. Von den gefilterten Prozessen wird der Name und die Speichermenge ausgegeben.

```
public static void Beispiel2()
{
    // LINQ-Abfrage
    var Prozesse =
        from p in System.Diagnostics.Process.GetProcesses()
        where p.WorkingSet64 < 700000
        select new { p.ProcessName, p.WorkingSet64 };

    // Nutzung des Abfrageergebnisses
    foreach (var Prozess in Prozesse)
    {
        Console.WriteLine(Prozess.ProcessName + ": " + Prozess.WorkingSet64);
    }
}
```

Listing 10.5 Filtern der Prozessliste (C#)

```
Public Sub Beispiel2()
    'LINQ-Abfrage
    Dim Prozesse =
        From p In System.Diagnostics.Process.GetProcesses() _
        Where (p.WorkingSet64 < 700000) _
        Select New With {p.ProcessName, p.WorkingSet64}

    ' Nutzung des Abfrageergebnisses
    Dim Prozess
    For Each Prozess In Prozesse
        Console.WriteLine(Prozess.ProcessName & ": " & Prozess.WorkingSet64)
    Next
End Sub
```

Listing 10.6 Filtern der Prozessliste (VB)

HINWEIS In dem zweiten Beispiel ist der Einsatz des Schlüsselwortes `var` anstelle eines konkreten Typnamens bzw. `Dim` ohne Datentyp zu beachten. Der Grund dafür ist, dass durch die Reduktion der Prozessliste auf die Attribute `ProcessName` und `WorkingSet64` ein anonymer Typ (vgl. Kapitel 6 »Sprachsyntax Visual Basic 2010 (VB.NET 10.0) und C# 2010 (C# 4.0)«) entsteht.

WICHTIG Es gibt drei wichtige Voraussetzungen, damit die LINQ-Abfragesyntax in MSIL (alias CIL) übersetzt werden kann:

- Es muss der Compiler für C# 3.0 oder höher bzw. Visual Basic 9.0 oder höher eingesetzt werden. Das heißt, die Syntax ist nicht in Projekten verfügbar, die als *Target Framework* .NET 2.0 oder 3.0 ausgewählt haben, sondern nur in .NET 3.5- oder 4.0-Projekten.
- Die Assembly *System.Core.dll* muss in dem Projekt referenziert sein
- Der Namensraum *System.Linq* muss importiert sein

Häufig wird Bedingung 3 übersehen. Dies erkennt man an der Fehlermeldung »Could not find an implementation of the query pattern for source type '...'«.

Da `select`, `where`, `from`, etc. ja Schlüsselwörter der Programmiersprachen C# und Visual Basic sind, stellt sich der kritische Leser sicherlich die Frage, warum Bedingungen 2 und 3 erfüllt sein müssen. Bisher gab es keine Schlüsselwörter, die von Referenzen und Importanweisungen abhängig waren. Der Grund liegt in diesem Fall darin, dass der Compiler die LINQ-Abfragesyntax in einem ersten Übersetzungsschritt in LINQ-Methodensyntax übersetzt. Diese Methoden sind Erweiterungsmethoden für bestehende Typen. Wenn diese Erweiterungsmethoden aber nicht verfügbar sind, schlägt die Übersetzung fehl.

LINQ-Methodensyntax

Wie bereits im vorangegangenen Abschnitt erwähnt, sind alle LINQ-Anweisungen intern als Methodenauf-rufe realisiert. So wird z. B. das Schlüsselwort `where` der Abfragesyntax auf die Erweiterungsmethode `Where()` abgebildet, `orderby` ist realisiert durch `OrderBy()` und `select` durch `Select()`. Durch die Aneinanderreihung der Methodenauf-rufe können komplexe Abfragen definiert werden.

Abfragesyntax	Methodensyntax
<pre>// LINQ-Abfrage in Abfragesyntax IEnumerable<string> Monate4 = from Monat in AlleMonate where Monat.Length == 4 orderby Monat select Monat.Substring(0, 3);</pre>	<pre>// LINQ-Abfrage in Methodensyntax IEnumerable<string> Monate4 = AlleMonate .Where(Monat => Monat.Length == 4) .OrderBy(Monat => Monat) .Select(Monat => Monat.Substring(0,3));</pre>

Tabelle 10.1 Vergleich von Abfragesyntax und Methodensyntax an einem Beispiel

Tatsächlich existiert nur für einen sehr kleinen Teil der Möglichkeiten von LINQ eine Repräsentation in der Abfragesyntax. Viele Möglichkeiten sind – insbesondere in C# – nur in der Methodensyntax verfügbar, z. B. `Top()`, `Skip()`, `Distinct()`, `Min()`, `Average()` etc.

Um die Monate 6 bis 8 in der Liste zu ermitteln, kann man mit `Skip()` die ersten fünf überspringen und dann mit `Take()` die nächsten drei auswählen.

```
// LINQ-Abfrage in Methodensyntax
IEnumerable<string> SommerMonate =
    AlleMonate
        .Select(Monat => Monat.Substring(0, 3))
        .Skip(5).Take(3);
```

Listing 10.7 Beispiel in Methodensyntax

Die Methodensyntax ist nicht so elegant wie die Abfragesyntax. Man kann aber die beiden Syntaxformen miteinander kombinieren, indem man den Ausdruck in Abfragesyntax in runden Klammern einschließt und auf diesem Ausdruck dann die Erweiterungsmethoden anwendet.

```
// LINQ-Abfrage in gemischter Syntax
IEnumerable<string> SommerMonate =
    (from Monat in AlleMonate
     select Monat.Substring(0, 3))
        .Skip(5).Take(3);
```

Listing 10.8 Beispiel in gemischter Syntax

HINWEIS In Visual Basic ist die Abfragesyntax umfangreicher als in C#. In C# kann man aber auch alle LINQ-Befehle nutzen, zum Teil ist die Anwendung aber wesentlich uneleganter als in Visual Basic.

Es gibt zur Laufzeit keinen Unterschied zwischen den beiden Syntaxformen. Auch die Mischung der Syntaxformen hat keinen Nachteil, denn die Klammerung sorgt nicht dafür, dass der Teilausdruck vorher ausgewertet wird. LINQ-Ausdrücke werden immer erst bei ihrer ersten Verwendung ausgeführt (Verzögerte Ausführung). Eine Ausnahme bilden die Konvertierungsmethoden `ToArray()`, `ToDictionary()`, `ToList()` und `ToLookup()`. Diese vier Methoden sorgen dafür, dass der davorstehende LINQ-Befehl sofort ausgeführt wird.

Übersicht über die LINQ-Befehle

Die folgende Tabelle zeigt die Liste aller in .NET 3.5/4.0 verfügbaren LINQ-Befehle. LINQ-Befehle werden auch *LINQ-Operatoren* genannt.

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
Aggregate			Eigene Aggregatfunktionen	–
All		Aggregate ... In ... Into All()	Liefert <i>true</i> , wenn alle Elemente einer Menge die angegebene Bedingung erfüllen	–
Any		Aggregate ... In ... Into Any()	Liefert <i>true</i> , wenn mindestens ein Element der Menge die angegebene Bedingung erfüllt	EXISTS
Average			Mittelwert (arithmetischer Durchschnitt)	AVG
Cast	from Typ x in Menge	From ... As ...	Typumwandlung aller Elemente der Menge	–
Concat			Vereinigungsmenge zweier Mengen	UNION
Contains			Prüft, ob die Menge ein bestimmtes Element enthält	IN
Count		Aggregate ... In ... Into Count()	Liefert die Anzahl der Elemente in der Menge in Form einer 32-Bit-Ganzzahl (Typ <code>Int32</code>)	COUNT
Distinct		Distinct	Entfernt alle doppelten Elemente in der Liste	DISTINCT
ElementAt			Liefert das Element in der Menge an einer bestimmten Stelle (Index)	–

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
ElementAtOrDefault			Liefert das Element in der Menge an einer bestimmten Stelle (Index) oder einen Standardwert, wenn der Index negativ oder größer als die Anzahl der Elemente ist	–
Empty			Erstellt eine leere Menge vom angegebenen Typ	–
Except			Vergleicht zwei Mengen und liefert nur diejenigen Elemente, die in der ersten Menge (die Menge, auf die die Methode angewendet wird), aber nicht in der zweiten Menge (die Menge, die als Parameter angegeben wird) vorhanden sind	–
First			Das erste Element einer Menge. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen. Wenn es kein Element gibt, tritt ein Laufzeitfehler auf.	–
FirstOrDefault			Das erste Element einer Menge oder ein Standardwert (bei Referenztypen <i>null</i> bzw. <i>Nothing</i>), wenn die Menge leer ist. Wenn mehrere Elemente in der Menge sind, werden alle anderen bis auf das erste verworfen.	–
GroupBy	group ... by ... into ...	Group ... By ... Into ...	Gruppieren eine Menge nach dem angegebenen Kriterium	GROUP BY
GroupJoin	join ... in ... on ... equals ... into ...	Group Join ... In ... On ...	Verbindet zwei Mengen durch einen OUTER JOIN	JOIN
Intersect			Liefert die Schnittmenge zweier Mengen	–
Join	join ... in ... on ... equals ...	Join ... In ... On ... Equals ...	Verbindet zwei Mengen durch einen INNER JOIN	JOIN
Last			Liefert das letzte Element einer Menge	–
LastOrDefault			Liefert das letzte Element einer Menge oder einen Standardwert, wenn die Menge leer ist	–



Methodenname	Schlüsselwort in der Abfrage-syntax (C#)	Schlüsselwort in der Abfrage-syntax (Visual Basic)	Beschreibung	Äquivalent in SQL
LongCount		Aggregate ... In ... Into LongCount()	Liefert die Anzahl der Elemente in der Menge in Form einer 64-Bit Ganzzahl (Typ Int64)	COUNT
Max		Aggregate ... In ... Into Max()	Ermittelt den maximalen Wert einer Menge	MAX
Min		Aggregate ... In ... Into Min()	Ermittelt den minimalen Wert einer Menge	MIN
OfType			Liefert alle Elemente einer Menge, die Instanzen einer bestimmten Klasse sind	–
OrderBy	orderby	Order By	Sortiert eine Menge aufsteigend	ORDER BY
OrderByDescending	OrderBy ... descending	Order By ... Descending	Sortiert eine Menge absteigend	ORDER BY DESC
Range			Erzeugt eine Menge mit den numerischen Werten von n bis m	–
Repeat			Erzeugt eine Menge mit n -Mal dem gleichen Element	–
Reverse			Umkehren der Reihenfolge	
Select	select	Select	Bestimmt die Daten und bildet die Elemente, die aus einer Menge erstellt werden	SELECT
SelectMany			Durchläuft Mengen, die selbst Mitglieder anderer Mengen sind und liefert eine flache Liste	–
SequenceEqual			Prüft, ob zwei Mengen identisch sind hinsichtlich der Anzahl, Reihenfolge und Inhalt der Elemente	–
Single			Das erste Element einer Menge. Wenn es kein Element gibt oder wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.	–

Methodenname	Schlüsselwort in der Abfragesyntax (C#)	Schlüsselwort in der Abfragesyntax (Visual Basic)	Beschreibung	Äquivalent in SQL
SingleOrDefault			Das erste Element einer Menge. Wenn es kein Element gibt, wird der Standardwerte (bei Referenztypen <i>null</i> oder <i>Nothing</i>) geliefert. Wenn mehrere Elemente in der Menge sind, tritt ein Laufzeitfehler auf.	–
Skip		Skip	Überspringt die ersten <i>n</i> Elemente einer Menge und liefert den Rest	–
SkipWhile		Skip While	Überspringt so lange Elemente, wie eine Bedingung erfüllt wird und liefert den Rest	–
Sum		Aggregate ... In ... Into Sum()	Summiert die Elemente einer Menge	SUM
Take		Take	Liefert die ersten <i>x</i> Elemente einer Menge	TOP
TakeWhile		Take While	Liefert so lange Elemente, wie eine Bedingung erfüllt wird	–
ThenBy	orderby ..., ...	Order By ..., ...	Angabe eines weiteren aufsteigenden Ordnungskriteriums bei einer Sortierung	ORDER BY
ThenByDescending	orderby ..., ... descending	Order By ..., ... Descending	Angabe eines weiteren absteigenden Ordnungskriteriums bei einer Sortierung	ORDER BY
ToArray			Konvertiert eine Menge zu einem Array	–
ToDictionary			Konvertiert eine Menge zu einer generischen Dictionary<K,T>-Menge	–
ToList			Konvertiert eine Menge zu einer generischen List<T>-Menge	–
ToLookup			Konvertiert eine Menge zu einer generischen Lookup<K,T>-Menge.	–
Union			Vereint zwei Mengen zu einer	UNION
Where	where	Where	Filtern der Eingabemenge	WHERE

Tabelle 10.2 LINQ-Befehle

Neben den LINQ-Befehlen kann man auch die Methoden der .NET-Klassenbibliothek in LINQ-Abfragen verwenden. Sinnvoll sind z. B. die Methoden der Klassen `System.String` (z. B. `StartsWith()`), `System.DateTime` (z. B. `AddYears()` und `System.Math` (z. B. `Round()`). Mit LINQ to Objects kann man prinzipiell alle Methoden der .NET Klassenbibliothek und auch eigene Methoden in eigenen Geschäftsobjekten nutzen. Mit anderen LINQ-Providern ist dies nur dann möglich, wenn es für die Methode eine Entsprechung in der Basissyntax gibt. Dies gilt bei LINQ to SQL im Wesentlichen nur für einige Methoden der Klassen `System.String`, `System.Math` und `System.DateTime`. Andere Methoden und selbstdefinierte Methoden haben keine Entsprechung in SQL und können daher auch nicht in LINQ to SQL genutzt werden.

ACHTUNG Ob die Reihenfolge der Befehle entscheidend ist, hängt von dem LINQ-Provider ab. Bei LINQ to Objects ist

```
from x in Zahlen where x < 50 orderby x select x
```

viel schneller als

```
from x in Zahlen orderby x where x < 50 select x
```

Bei LINQ to Entities gibt es keinen Unterschied, denn die zugrundeliegende Datenbank wird dies optimieren.

LINQ to Objects

Mit LINQ to Objects wird die Abfrage von Objektmengen im Hauptspeicher bezeichnet. Abgefragt werden können alle Objektmengen, die entweder die Schnittstelle `IEnumerable` oder ihr generisches Pendant `IEnumerable<T>` unterstützen. Dies sind also die Klassen in `System.Collections` (z. B. `ArrayList`, `Hashtable`, `Queue` und `Stack`), die Klassen in `System.Collections.Generic` (z. B. `List<T>`, `SortedDictionary<T>`, `Queue<T>` und `Stack<T>`), die Klasse `System.Array` sowie spezielle Mengen wie `DataRowCollection`, `DataColumnCollection`, `DirectoryEntries` und `ManagementObjectCollection`. Da `IEnumerable` bzw. `IEnumerable<T>` Voraussetzung für das Funktionieren der `foreach`-Schleife sind, besitzt praktisch jede Menge in der .NET-Klassenbibliothek eine der beiden Schnittstellen. Für LINQ to Objects ist es unerheblich, ob die Menge vom .NET Framework erzeugt wird oder von eigenem Programmcode.

LINQ to Objects mit elementaren Datentypen

Am Beispiel einer Menge von Zahlen in Form eines Arrays vom Typ `Int32` soll die Anwendung von LINQ-Befehlen auf elementaren Datentypen gezeigt werden.

Gegeben sind zwei Zahlenmengen:

```
int[] Zahlen1 = { 15, 4, 11, 3, 19, 8, 16, 7, 12, 5, 9, 20, 1, 4, 8, 13, 14, 4, 1 };  
int[] Zahlen2 = { 12, 5, 31, 24, 29, 20, 13, 31 };
```

Listing 10.9 Definition der Zahlenmenge

Das folgende Listing enthält zahlreiche Fragestellungen in Bezug auf diese beiden Zahlenmengen und den Weg, die Lösung mit LINQ zu ermitteln. Das jeweilige Ergebnis wird aus Platzgründen hier nicht abgedruckt. Durch den Programmcode zu diesem Buch können Sie dies jedoch selbst ausprobieren.

```
private static void Demo_LTO_Zahlen()
{
    int i;
    double d;

    string s = "Geben Sie die Zahlen aus, die kleiner als 10 sind.";
    var Ergebnis =
        from n in Zahlen1
        where n < 10
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie die Zahlen, die kleiner als 10 sind, aufsteigend sortiert aus.";
    Ergebnis =
        from n in Zahlen1
        where n < 10
        orderby n // optional
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie die Zahlen, die kleiner als 10 sind, absteigend sortiert aus.";
    Ergebnis =
        from n in Zahlen1
        where n < 10
        orderby n descending
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie die Zahlen, die kleiner als 10 sind, absteigend sortiert aus." +
        "Eliminieren Sie alle Duplikate.";
    Ergebnis =
        (from n in Zahlen1
         where n < 10
         orderby n descending
         select n).Distinct();
    Print(Ergebnis, s);

    s = "Geben Sie die vierte bis achte Zahl aus.";
    Ergebnis =
        (from n in Zahlen1
         where n < 10
         select n).Skip(3).Take(4);
    Print(Ergebnis, s);

    s = "Geben Sie die erste Zahl aus!";
    i =
        (from n in Zahlen1
         select n).First();
    Print(i, s);

    s = "Geben Sie die letzte Zahl aus!";
    i =
        (from n in Zahlen1
         select n).Last();
    Print(i, s);
}
```

```
s = "Geben Sie die 10. Zahl aus!";
i =
    (from n in Zahlen1
     select n).ElementAt(9);
Print(i, s);

s = "Geben Sie die 50. Zahl aus! (Fangen Sie den Fehler ab!)";
i =
    (from n in Zahlen1
     select n).ElementAtOrDefault(49);
Print(i, s);

s = "Geben Sie die Anzahl der Zahlen aus.";
i =
    (from n in Zahlen1
     select n).Count();
Print(i, s);

s = "Geben Sie nur die niedrigste Zahl aus.";
i =
    (from n in Zahlen1
     select n).Min();
Print(i, s);

s = "Geben Sie nur die höchste Zahl aus.";
i =
    (from n in Zahlen1
     select n).Max();
Print(i, s);

s = "Geben Sie den Durchschnitt aus.";
d =
    (from n in Zahlen1
     select n).Average();
Print(d, s);

s = "Geben Sie die Summe aus.";
d =
    (from n in Zahlen1
     select n).Sum();
Print(d, s);

s = "Geben Sie das Produkt aller Werte aus.";
d =
    (from n in Zahlen1
     select n).Aggregate((summe, wert) => summe * wert);
Print(d, s);

s = "Gruppieren Sie die Werte.";
IEnumerable<IGrouping<int, int>> GruppeErgebnis =
    (from n in Zahlen1
     group n by n);
Print(GruppeErgebnis, s);
```



```
s = "Geben Sie die Häufigkeit eines jeden Werts aus!";
IDictionary<int, int> GruppeHaeufigkeit =
    (from n in Zahlen1
     group n by n into g
     select new { Wert = g.Key, Anzahl = g.Count() }
     ).ToDictionary(y => y.Wert, y => y.Anzahl);
Print(GruppeHaeufigkeit, s);

s = "Verbinden Sie die Zahlenmengen 1 und 2:";
Ergebnis = (from n in Zahlen1 select n).Union(from n2 in Zahlen2 select n2);
Print(Ergebnis, s);

s = "Verbinden Sie die Zahlenmengen 1 und 2 und sortieren Sie das Ergebnis:";
Ergebnis = (from n in Zahlen1 select n).Union(from n2 in Zahlen2 select n2).OrderBy(n => n);
Print(Ergebnis, s);

s = "Bilden Sie die Schnittmenge aus den Zahlenmengen 1 und 2.";
Ergebnis = (from n in Zahlen1 select n).Intersect(from n2 in Zahlen2 select n2).OrderBy(n => n);
Print(Ergebnis, s);

s = "Schließen Sie die Zahlen aus Zahlenmengen 2 in Menge 1 aus.";
Ergebnis = (from n in Zahlen1 select n).Except(from n2 in Zahlen2 select n2).OrderBy(n => n);
Print(Ergebnis, s);

s = "Prüfen Sie, ob die Zahlenmenge 1 und 2 die gleichen Zahlen in der gleichen Reihenfolge enthalten.";
bool Erfuehlt = (from n in Zahlen1 select n).SequenceEqual(from n2 in Zahlen2 select n2);
Print(Erfuehlt, s);

s = "Prüfen Sie, ob die Zahl 20 in der Menge vorkommt.";
Erfuehlt =
    (from n in Zahlen1
     orderby n descending
     select n).Contains(14);
Print(Erfuehlt, s);

s = "Prüfen Sie, ob Zahlen größer als 20 in der Menge vorkommen.";
Erfuehlt =
    (from n in Zahlen1
     orderby n descending
     select n).Any(n => n > 20);
Print(Erfuehlt, s);

s = "Prüfen Sie, ob alle Zahlen kleiner 20 sind.";
Erfuehlt =
    (from n in Zahlen1
     orderby n descending
     select n).All(n => n < 20);
Print(Erfuehlt, s);

s = "Filtern Sie alle Integer-Werte heraus!";
Ergebnis =
    (from n in Zahlen1 select n).OfType<int>();

Print(Ergebnis, s);
```

```

s = "Wandeln Sie alle Zahlen in Byte-Werte um!";
var kleineZahlen =
    (from n in Zahlen1 select n).Cast<byte>();
foreach (var x in kleineZahlen)
{
    Console.WriteLine(x);
}
Print(kleineZahlen, s);
}

```

Listing 10.10 Anwendungsbeispiele von LINQ to Objects auf Zahlenmengen

Das obige Listing nutzt zur Ausgabe die selbstdefinierte Methode Print(). Es muss aber mehrere Überladungen von Print() geben, da die LINQ-Abfragen unterschiedliche Ergebnisse liefern können:

- Viele der obigen LINQ-Abfragen liefern wieder eine Zahlenmenge zurück. Der konkrete Datentyp, der zurückgeliefert wird, ist von den eingesetzten Methoden abhängig. Alle diese Klassen besitzen jedoch die Schnittstelle `IEnumerable<int>`. Zum Durchlaufen des Ergebnisses ist eine einfache Schleife ausreichend.
- Durch das Gruppieren von Elementen ohne das Schlüsselwort `into` entstehen zwei verschachtelte Objektmengen des Typs `IEnumerable<IGrouping<int, int>>`. Die obere Menge repräsentiert dabei die Gruppen, die untergeordnete Menge die Elemente in jeder Gruppe. Zum Durchlaufen des Ergebnisses ist eine geschachtelte Schleife notwendig. Diese Form des Gruppierens bezeichnet man als hierarchisches Gruppieren.
- Durch das Gruppieren von Elementen mit dem Schlüsselwort `into` entsteht ein neuer anonymer Typ, der das Gruppierungskriterium und die zusammengefassten Daten anderer Mitglieder des Ausgangstyps enthält. Das Ergebnis ist ein Dictionary-Objekt mit zwei `Int32`-Werten: `IDictionary<int, int>`. Diese Form des Gruppierens entspricht dem flachen Gruppieren aus SQL. Trotz der Verwendung von `into` kann man hierarchisches Gruppieren erreichen, wenn man in dem anonymen Typ auf die Gruppe selbst verweist, z. B. `from p in System.Diagnostics.Process.GetProcesses group p by p.ProcessName into g select new { Name = g.Key, Anzahl = g.Count(), Max = g.Max(p => p.WorkingSet64), ProzesseInDieserGruppe = g };`

```

private static void Print(IEnumerable<int> Nums, string s)
{
    HeadLine(s);
    foreach (int x in Nums)
    {
        Console.WriteLine(x);
    }
}

private static void Print(IDictionary<int, int> gruppe, string s)
{
    HeadLine(s);
    foreach (var x in gruppe)
    {
        Console.WriteLine(x.Key + ": " + x.Value);
    }
}

```

```
private static void Print(IEnumerable<IGrouping<int, int>> Gruppen, string s)
{
    HeadLine(s);
    foreach (IGrouping<int, int> x in Gruppen)
    {
        Console.WriteLine("---- " + x.Key);
        foreach (int i in x)
        {
            Console.WriteLine(i);
        }
    }
}
```

Listing 10.11 Ausgaberroutinen für die Ergebnisse der LINQ-Abfragen (Auswahl)

LINQ to Objects mit komplexen Typen des .NET Framework

Die Anwendung von LINQ to Objects auf komplexe Datentypen unterscheidet sich von der Anwendung auf elementare Datentypen wie folgt:

- Bei LINQ to Objects mit elementaren Datentypen wurde die in dem from-Ausdruck deklarierte Laufvariable selbst für Bedingungen, Sortierungen und Berechnungen verwendet. Bei komplexen Datentypen muss mithilfe der Laufvariablen Bezug auf ein Mitglied des Objekts genommen werden.
- LINQ to Objects mit elementaren Datentypen liefert in der Regel eine Menge des Eingabetyps zurück. Bei komplexen Datentypen kann alternativ ein anonymer Typ zurückgegeben werden, der nur eine Teilmenge der Mitglieder des Ausgangstyps enthält. Dies nennt man eine Projektion.

Beispiel

In dem folgenden Beispiel werden LINQ-Befehle auf einer Menge von Objekten des Typs `System.Diagnostics.Process` angewendet. Die statische Methode `GetProcesses()` der Klasse `System.Diagnostics.Process` liefert eine Liste der laufenden Prozesse auf einem System in Form eines Arrays mit Instanzen von `System.Diagnostics.Process`.

```
private static void Demo_LT0_Prozesse()
{
    Process[] Prozesse = Process.GetProcesses();

    Process p;
    long i;
    double d;

    string s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes Speicher verbrauchen.";
    var Ergebnis =
        from n in Prozesse
        where n.WorkingSet64 < 3000000
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes Speicher verbrauchen. Sortieren Sie die Liste aufsteigend nach Speicherverbrauch.";
    Ergebnis =
        from n in Prozesse
```

```

where n.WorkingSet64 < 3000000
orderby n.WorkingSet64 // optional
select n;
Print(Ergebnis, s);

```

s = "Geben Sie alle Prozesse aus, die weniger als 3.000.000 Bytes Speicher verbrauchen. Sortieren Sie die Liste absteigend nach Speicherverbrauch.";

```

Ergebnis =
from n in Prozesse
where n.WorkingSet64 < 3000000
orderby n.WorkingSet64 descending // optional
select n;
Print(Ergebnis, s);

```

s = "Geben Sie die Prozesse aus. Eliminieren Sie alle Duplikate.";

```

Ergebnis =
    (from n in Prozesse
     select n).Distinct();
Print(Ergebnis, s);

```

s = "Geben Sie den vierten bis achten Prozess aus in der nach Speicherverbrauch aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes verbrauchen.";

```

Ergebnis =
    (from n in Prozesse
     where n.WorkingSet64 > 1000000
     orderby n.WorkingSet64
     select n).Skip(3).Take(4);
Print(Ergebnis, s);

```

s = "Geben Sie den ersten Prozess aus in der nach Speicherverbrauch aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes verbrauchen.";

```

p =
    (from n in Prozesse
     where n.WorkingSet64 > 1000000
     orderby n.WorkingSet64
     select n).First();
Print(p, s);

```

s = "Geben Sie den letzten Prozess aus in der nach Speicherverbrauch aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes verbrauchen.";

```

p =
    (from n in Prozesse
     where n.WorkingSet64 > 1000000
     orderby n.WorkingSet64
     select n).Last();
Print(p, s);

```

s = "Geben Sie den 10. Prozess aus in der nach Speicherverbrauch aufsteigend sortierten Liste aller Prozesse, die mehr als 1.000.000 Bytes verbrauchen.";

```

p =
    (from n in Prozesse
     where n.WorkingSet64 > 1000000
     orderby n.WorkingSet64
     select n).ElementAt(9);
Print(p, s);

```

s = "Geben Sie den 150. Prozess aus! (Fangen Sie den Fehler ab!)"

```

p =
    (from n in Prozesse
     select n).ElementAtOrDefault(149);
Print(p, s);

```

```

s = "Geben Sie die Anzahl der Prozesse aus (mit einem LINQ-Statement!)";
i =
    (from n in Prozesse
     select n).Count();
Print(i, s);

s = "Geben Sie nur den niedrigsten Speicherverbrauch aus";
i =
    (from n in Prozesse
     select n).Min(n => n.WorkingSet64);
Print(i, s);

s = "Geben Sie nur den höchsten Speicherverbrauch aus";
i =
    (from n in Prozesse
     select n).Max(n => n.WorkingSet64);
Print(i, s);

s = "Geben Sie den durchschnittlichen Speicherverbrauch aus";
d =
    (from n in Prozesse
     select n).Average(n => n.WorkingSet64);
Print(i, s);

s = "Geben Sie die Summe des Speicherverbrauchs aus";
i =
    (from n in Prozesse
     select n).Sum(n => n.WorkingSet64);
Print(i, s);

s = "Gruppieren Sie die Prozesse nach Namen.";
IEnumerable<IGrouping<string, Process>> GruppeErgebnis =
    (from n in Prozesse
     group n by n.ProcessName);
Print(GruppeErgebnis, s);

s = "Geben Sie die Häufigkeit eines jeden Prozessnamens aus!";
IDictionary<string, int> GruppeHaeufigkeit =
    (from n in Prozesse
     group n by n.ProcessName into g
     select new { Name = g.Key, AnzProzess = g.Count() }
     ).ToDictionary(y => y.Name, y => y.AnzProzess);
Print(GruppeHaeufigkeit, s);

s = "Starten Sie einen neuen Prozess (Notepad) und ermitteln Sie, durch einen Vergleich der
Prozessliste vorher und nachher, welche Prozesse neu hinzugekommen sind. (Geben Sie die Process-ID und
den Prozessnamen aus!);
Process neupro = Process.Start(@"C:\Windows\notepad.exe");
neupro.WaitForInputIdle();
Process[] Prozesse2 = Process.GetProcesses();

//Print((from p1 in Prozesse where p1.ProcessName=="notepad" select p1), "Test");
//Print((from p2 in Prozesse2 where p2.ProcessName=="notepad" select p2), "Test");

```

```

IEnumerable<int> ProzessListe = (from n2 in Prozesse2 select n2.Id).Except(from n in Prozesse select
n.Id);
Print(ProzessListe, s);

//var ProzessListe2 = from p in System.Diagnostics.Process.GetProcesses() select p.ProcessName;

s = "Listen Sie die Prozesse mit ihren Threads auf.";
var ProzesseMitThreads =
    (from n in Prozesse
     select new { n, n.Threads }
    );
HeadLine(s);
foreach (var x in ProzesseMitThreads)
{
    Console.WriteLine(x.n);
    try
    {
        foreach (ProcessThread y in x.Threads)
        {
            Console.WriteLine(y.StartTime);
        }
    }
    catch (Exception)
    {
    }
}

s = "Geben Sie zu jedem Prozess die Anzahl der Threads aus!";
var ProzesseMitThreadCount =
    (from n in Prozesse
     where n.Id > zehn
     select new { n, n.Threads.Count }
    );
HeadLine(s);
foreach (var m in ProzesseMitThreadCount)
{
    Console.WriteLine(m.n + ":" + m.Count);
}

s = "Geben Sie die Prozesse aus, die mehr als 10 Threads haben!";
Ergebnis =
    (from n in Prozesse
     where n.Threads.Count > 10
     select n);
Print(Ergebnis, s);

s = "Geben Sie den/die Prozess(e) aus, der/die die meisten Threads hat!";
Ergebnis = (from n in Prozesse where n.Threads.Count == Prozesse.Max(x => x.Threads.Count) select n);
Print(Ergebnis, s);
}

```

Listing 10.12 Anwendungsbeispiele von LINQ to Objects auf eine Menge von Objekten des Typs System.Diagnostics.Process

LINQ to Objects mit eigenen Geschäftsobjekten

LINQ-Abfragen können auch über eigene (Geschäfts-)Objektmenge(n) gestellt werden, egal ob diese direkt durch Implementierung von `IEnumerable`/`IEnumerable<T>` oder durch Ableiten von einer der vordefinierten Mengenklassen implementiert wurden. Das folgende Objektmodell zeigt drei Mengen (`FlugMenge`, `PassagierMenge` und `BuchungsMenge`), die jeweils durch Ableiten von der Klasse `System.Collections.Generic.List<T>` realisiert wurden.

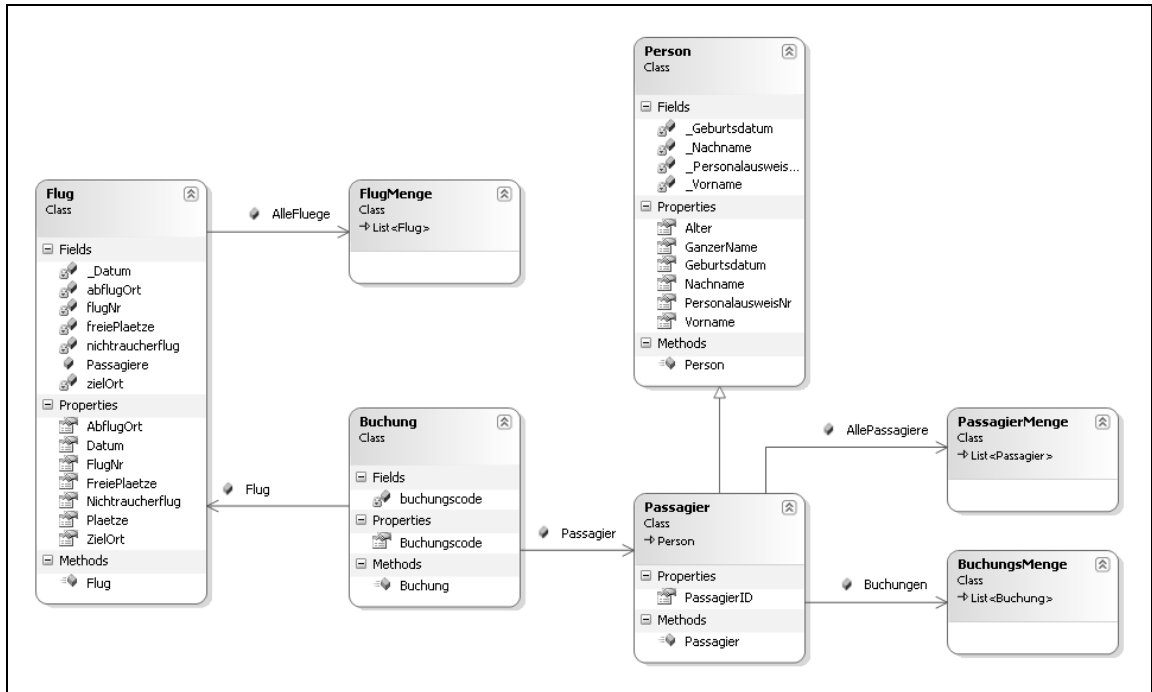


Abbildung 10.2 Objektmodell für die folgenden Beispiele

Beispiel

Das folgende Listing zeigt zahlreiche Beispiele zur Abfrage der Mengen in dem oben dargestellten Objektmodell. Das Listing setzt voraus, dass die Mengen vorher mit Daten gefüllt wurden. Diese Befüllung wird hier aus Platzgründen nicht abgedruckt, ist jedoch in den Codebeispielen zu diesem Buch enthalten.

```
private static void Demo_LTO_Objektmodell()
{
    // Initialisiere das Objektmodell
    BO Init.Init();
    string s;
    long i;
    Flug flug;
    double d;

    s = "Geben Sie alle Flüge von Rom abgehend aus!";
    var Ergebnis =
        from f in Flug.AlleFluege
        where f.AbflugOrt == "Rom"
```

```

    select f;
    Print(Ergebnis, s);

    s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben.";
    Ergebnis =
        from n in Flug.AlleFluege
        where n.FreiePlaetze < 100
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben. Sortieren Sie die Liste
aufsteigend nach Platzanzahl.";
    Ergebnis =
        from n in Flug.AlleFluege
        where n.FreiePlaetze < 100
        orderby n.FreiePlaetze
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie alle Flüge aus, die weniger als 100 freie Plätze haben. Sortieren Sie die Liste
absteigend nach Platzanzahl.";
    Ergebnis =
        from n in Flug.AlleFluege
        where n.FreiePlaetze < 100
        orderby n.FreiePlaetze descending
        select n;
    Print(Ergebnis, s);

    s = "Geben Sie Flug 101 aus.";
    flug = (from f in Flug.AlleFluege
            where f.FlugNr == 101
            select f).SingleOrDefault();
    Print(flug, s);

    s = "Geben Sie die Flüge aus, aber jede Strecke nur einmal!";
    var Strecken =
        (from n in Flug.AlleFluege
         select new { n.AbflugOrt, n.ZielOrt }).Distinct();
    HeadLine(s);

    foreach (var f in Strecken)
    {
        Console.WriteLine(f.AbflugOrt + " -> " + f.ZielOrt);
    }

    s = "Geben Sie alle Ziele aus, die von Rom aus erreichbar sind.";
    var Ziele =
        (from n in Flug.AlleFluege
         where n.AbflugOrt == "Rom"
         select n.ZielOrt).Distinct();
    HeadLine(s);

    foreach (string f in Ziele)
    {
        Console.WriteLine(f);
    }

    s = "Geben Sie den vierten bis achten Flug aus in der nach freien Plätzen aufsteigend sortierten
Liste aller Flüge, die in Berlin landen.";
    Ergebnis =
        (from n in Flug.AlleFluege
         where n.ZielOrt == "Berlin"

```



```
        orderby n.FreiePlaetze
        select n).Skip(3).Take(4);
Print(Ergebnis, s);

s = "Geben Sie den ersten Flug aus in der nach freien Plätzen aufsteigend sortierten Liste aller
Flüge, die in Berlin landen.";
flug =
    (from n in Flug.AlleFluege
     where n.ZielOrt == "Berlin"
     orderby n.FreiePlaetze
     select n).First();
Print(flug, s);

s = "Geben Sie den letzten Flug aus in der nach freien Plätzen aufsteigend sortierten Liste aller
Flüge, die in Berlin landen.";
flug =
    (from n in Flug.AlleFluege
     where n.ZielOrt == "Berlin"
     orderby n.FreiePlaetze
     select n).Last();
Print(flug, s);

s = "Geben Sie den 10. Flug aus in der nach freien Plätzen aufsteigend sortierten Liste aller Flüge,
die in Berlin landen.";
flug =
    (from n in Flug.AlleFluege
     where n.ZielOrt == "Berlin"
     orderby n.FreiePlaetze
     select n).ElementAt(9);
Print(flug, s);

s = "Geben Sie den 150. Flug aus in der nach freien Plätzen aufsteigend sortierten Liste aller Flüge,
die in Berlin landen.";
flug =
    (from n in Flug.AlleFluege
     where n.ZielOrt == "Berlin"
     orderby n.FreiePlaetze
     select n).ElementAtOrDefault(149);
Print(flug, s);

s = "Geben Sie Anzahl der Flüge aus (mit einem LINQ-Statement!)";
i =
    (from n in Flug.AlleFluege
     select n).Count();
Print(i, s);

s = "Geben Sie die geringste freie Platzanzahl aus.";
i =
    (from n in Flug.AlleFluege
     select n).Min(n => n.FreiePlaetze);
Print(i, s);

s = "Geben Sie die höchste freie Platzanzahl aus.";
i =
    (from n in Flug.AlleFluege
     select n).Max(n => n.FreiePlaetze);
Print(i, s);
```

```

s = "Geben Sie die durchschnittliche freie Platzanzahl aus.";
d =
    (from n in Flug.AlleFluege
     select n).Average(n => n.FreiePlaetze);
Print(d, s);

s = "Geben Sie Summe aller freien Plätze aus.";
i =
    (from n in Flug.AlleFluege
     select n).Sum(n => n.FreiePlaetze);
Print(i, s);

s = "Gruppieren Sie die Flüge nach Abflugorten.";
IEnumerable<IGrouping<string, Flug>> GruppeErgebnis =
    (from n in Flug.AlleFluege
     group n by n.AbflugOrt);
Print(GruppeErgebnis, s);

s = "Geben Sie die Häufigkeit eines jeden Abflugortes aus!";
IDictionary<string, int> GruppeHaeufigkeit =
    (from n in Flug.AlleFluege
     group n by n.AbflugOrt into g
     select new { Name = g.Key, AnzFlug = g.Count() }
     ).ToDictionary(y => y.Name, y => y.AnzFlug);
Print(GruppeHaeufigkeit, s);

s = "Erstellen Sie eine gruppierte Liste aller Passagiere mit ihren Buchungen!";
var pass2 = from p in Passagier.AllePassagiere
            orderby p.GanzerName
            select new { p.GanzerName, p.Buchungen };
foreach (var p in pass2)
{
    Console.WriteLine(p.GanzerName);
    foreach (Buchung b in p.Buchungen)
        Console.WriteLine("\t" + b.Buchungscode);
}

s = "Erstellen Sie die Liste der zehn Passagiere mit den meisten Buchungen.";
var pass = (from p in Passagier.AllePassagiere
            orderby p.Buchungen.Count descending
            select p).Take(10);
Print(pass, s);

s = "Erstellen Sie die Liste des/der Passagier(e) mit den meisten Buchungen.";
pass = (from n in Passagier.AllePassagiere where n.Buchungen.Count == Passagier.AllePassagiere.Max(x
=> x.Buchungen.Count) select n);
Print(pass, s);

s = "Finden Sie alle Passagiere, die nach Rom fliegen.";
pass = (from p in Passagier.AllePassagiere
        where p.Buchungen.Any(b => b.Flug.ZielOrt == "Rom")
        select p);
Print(pass, s);

```

```

s = "Finden Sie alle Passagiere, die genauso viele Buchungen haben wie ein Flug freie Plätze.";
var joinpass = (from p in Passagier.AllePassagiere
                join f in Flug.AlleFluege
                on p.Buchungen.Count equals f.FreiePlaetze
                select new { p.GanzerName, f.FlugNr, p.Buchungen.Count, f.FreiePlaetze });
HeadLine(s);
foreach (var j in joinpass)
{
    Console.WriteLine(j.GanzerName + " und Flug " + j.FlugNr + " haben die gleiche Zahl: " + j.Count + "
/ " + j.FreiePlaetze);
}

s = "Geben Sie alle Passagiere aus und optional dazu einen Flug, der genausoviele freie Plätze hat
wie der Passagier Buchungen hat.";
var joinpass2 = (from p in Passagier.AllePassagiere
                 join f in Flug.AlleFluege
                 on p.Buchungen.Count equals f.FreiePlaetze
                 into Fluege
                 select new { p.GanzerName, Fluege });
HeadLine(s);
foreach (var j in joinpass2)
{
    Console.WriteLine(j.GanzerName + " hat " + j.Fluege.Count() + " korrespondierende Flüge!");
}

s = "Geben Sie alle Passagiere aus, die älter als 50 Jahre sind!";
pass = (from p in Passagier.AllePassagiere
        where p.Geburtsdatum.AddYears(50) < DateTime.Now
        select p);
Print(pass, s);

s = "Geben Sie alle Flüge aus, mit Passagieren älter als 50 Jahre !";
Ergebnis = (from p in Passagier.AllePassagiere
             where p.Geburtsdatum.AddYears(50) < DateTime.Now
             from b in p.Buchungen
             select b.Flug).Distinct();
Print(Ergebnis, s);
}

```

Listing 10.13 Anwendungsbeispiele von LINQ to Objects auf verschiedene selbstdefinierte Geschäftsobjektmen-gen

Parallel LINQ (PLINQ)

Parallel LINQ (PLINQ, früher auch LINQ to Parallel) ist neu ab .NET 4.0. Es ermöglicht die Parallelisierung von LINQ to Objects-Abfragen auf mehrere Prozessoren/Prozessorkerne. Dadurch kann (!) sich eine Beschleunigung ergeben.

PLINQ ist realisiert in Form der Erweiterungsmethode `AsParallel()`, die auf einfache Weise in LINQ to Objects-Abfragen integriert werden kann.

Das folgende Beispiel zeigt eine einfache Abfrage mit Filtern (`where`) und Sortieren (`orderby`) über eine Zahlenreihe mit Einsatz von `AsParallel()`.

```

/// <summary>
/// Massendaten filtern und sortieren mit PLINQ
/// </summary>
public static void LTOMassendaten_mit_PLINQ()
{
    long AnzZahlen = 1000000;
    System.Random rnd = new Random(DateTime.Now.Year);
    List<long> Zahlen = new List<long>();
    for (int i = 1; i <= AnzZahlen; i++) Zahlen.Add(rnd.Next(100));

    long Summe = 0;
    Stopwatch t = new Stopwatch();
    t.Start();
    for (int w = 1; w <= 20; w++)
    {
        var q = (from x in Zahlen.AsParallel() where x < 50 orderby x select x).ToList();
        Summe += q.Count();
    }
    t.Stop();
    Console.WriteLine("Summe: " + Summe);
    Console.WriteLine("Mit PLINQ = " + t.ElapsedMilliseconds);
}

```

Listing 10.14 Eine Abfrage ohne und mit PLINQ

Die folgende Tabelle zeigt Messergebnisse, auch im Vergleich, wenn man `AsParallel()` weglassen würde.

Anzahl Zahlen	Ohne PLINQ – ohne <code>AsParallel()</code>	Mit PLINQ – mit <code>AsParallel()</code>
10000	50 Millisekunden	76 Millisekunden
100000	441 Millisekunden	190 Millisekunden
1000000	5132 Millisekunden	1532 Millisekunden

Tabelle 10.3 Ausführungsdauer von LINQ to Objects ohne und mit PLINQ, jeweils auf dem gleichen Rechner mit Intel Core i7 mit acht Prozessorkernen

ACHTUNG Man sieht: Erst bei größeren Grundmengen lohnt der mit der Parallelisierung verbundene Zusatzaufwand!

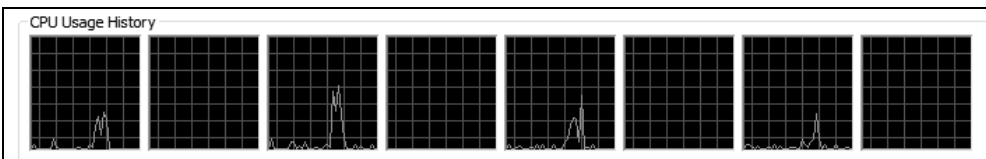


Abbildung 10.3 Auslastung von acht Kernen bei einer Abfrage ohne PLINQ

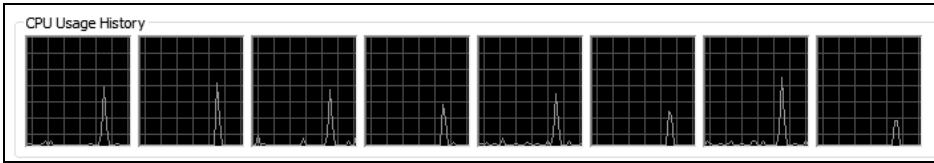


Abbildung 10.4 Auslastung von acht Kernen bei einer Abfrage mit PLINQ

ACHTUNG PLINQ bessert auch Reihenfolgefehler aus. Dort ist

```
from x in Zahlen orderby x where x < 50 select x
```

genauso schnell wie

```
from x in Zahlen where x < 50 orderby x select x
```

Ohne PLINQ dauert die erste LINQ to Objects-Abfrage bei 10000 Zahlen etwa doppelt so lange wie die zweite!

TIPP Bei Bedarf kann das Verhalten von PLINQ durch den Einsatz weiterer Erweiterungsmethoden beeinflusst werden. Wird zum Beispiel mit `AsOrdered()` festgelegt, dass die Sortierreihenfolge aus der Quelle erhalten bleiben soll, bringt dies im Zuge einer parallelen Abfrage etwas Mehraufwand mit sich und muss deswegen mit dieser Methode bei Bedarf angefordert werden. Mittels `WithCancellation()` wird darüber hinaus ein `CancellationToken` an die Abfrage übergeben, sodass deren Ausführung später abgebrochen werden kann. `WithDegreeOfParallelism()` gibt an, wie viele Tasks maximal für diese Anfrage verwendet werden dürfen. Standardmäßig werden so viele Tasks wie Kerne verwendet, die dann im Idealfall alle genutzt werden können. Kommt PLINQ zur Entscheidung, dass das Parallelisieren einer Abfrage nicht sinnvoll ist, so wird diese sequenziell ausgeführt. Dieses Verhalten kann allerdings mittels `WithExecutionMode()` beeinflusst werden. Im betrachteten Listing wird damit beispielsweise eine Parallelisierung erzwungen. Die letzte der verwendeten Optionen, `WithMergeOptions()`, legt fest, wie die Ergebnisse der unterschiedlichen Tasks kombiniert werden sollen. Mit `FullyBuffered` wird zum Beispiel erreicht, dass jeder Task sämtliche Ergebnisse in einen eigenen Buffer ablegt, wobei diese erst zum Schluss zur Ergebnismenge zusammengefügt werden.

Lesen Sie unbedingt »When To Use Parallel.ForEach and When to Use PLINQ?« [MSDN39], sowie weitere Artikel der Website »Parallel Computing with Managed Code« [MSDN40].

Leider kann der Inhalt hier aus Platzgründen nicht wiedergegeben werden.

LINQ to XML

Die Abfrage von XML-Dokumenten mit LINQ (LINQ to XML) wird im Rahmen des Kapitels zur XML-Verarbeitung mit .NET (Kapitel 13 »Datenzugriff mit System.Xml und LINQ to XML«) behandelt.

LINQ to DataSet

Die Abfrage von ADO.NET-Datasets wird im Kapitel 11 zu ADO.NET (»Datenzugriff mit ADO.NET«) behandelt.

LINQ to SQL

LINQ to SQL ist eine LINQ-Variante für das Objektrelational-Mapping (ORM) und die Umwandlung von LINQ, die eine extrem kurze Relevanzdauer hatte. LINQ to SQL erschien im November 2008 mit .NET 3.5 (als reine Lösung für Microsoft SQL Server) und wurde in .NET 3.5 SP 1 (August 2009) durch datenbankneutrale LINQ to Entities abgelöst. Es gibt LINQ to SQL zwar weiterhin, es gibt aber keinen Grund mehr, damit heute noch ein Projekt zu beginnen. LINQ to SQL wird in dieser Auflage des Buchs daher nicht mehr behandelt.

LINQ to Entities

LINQ to Entities ist in diesem Buch ein eigenes Hauptkapitel gewidmet: Kapitel 12 »Objektrelationales Mapping (ORM) mit dem ADO.NET Entity Framework 4.0«.

LINQ to DataServices

LINQ to DataServices wird im Kapitel 14 »Windows Communication Foundation (WCF) 4.0«, Abschnitt »WCF Data Service«.