C# und .NET 8

Grundlagen, Profiwissen und Rezepte

DIE LESEPROBE

7 Einführung in LINQ

LINQ (Language Integrated Query) ist ein C#-Sprachfeature, das mit C# 3.0 eingeführt wurde. Bei *LINQ to Objects* handelt es sich um die allgemeinste und grundlegendste LINQ-Implementierung, die auch die wichtigsten Bausteine für die übrigen LINQ-Implementierungen liefert. In einer SQL-ähnlichen Syntax können miteinander verknüpfte Collections/Auflistungen abgefragt werden, die die *IEnumerable*-Schnittstelle implementieren.

7.1 LINQ-Grundlagen

Der wichtigste Vorteil von LINQ ist die standardisierte Möglichkeit, nicht nur Tabellen in einer relationalen Datenbank, sondern auch Textdateien, XML-Dateien und andere Datenquellen in einer SQL-ähnlichen Syntax abzufragen. Ein zweiter Vorteil ist die Fähigkeit, diese standardisierten Methoden von jeder .NET-konformen Sprache (wie zum Beispiel C# oder VB) aus aufrufen zu können.

7.1.1 Die LINQ-Architektur

Die folgende Abbildung soll die grundsätzliche Architektur von LINQ verdeutlichen.

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

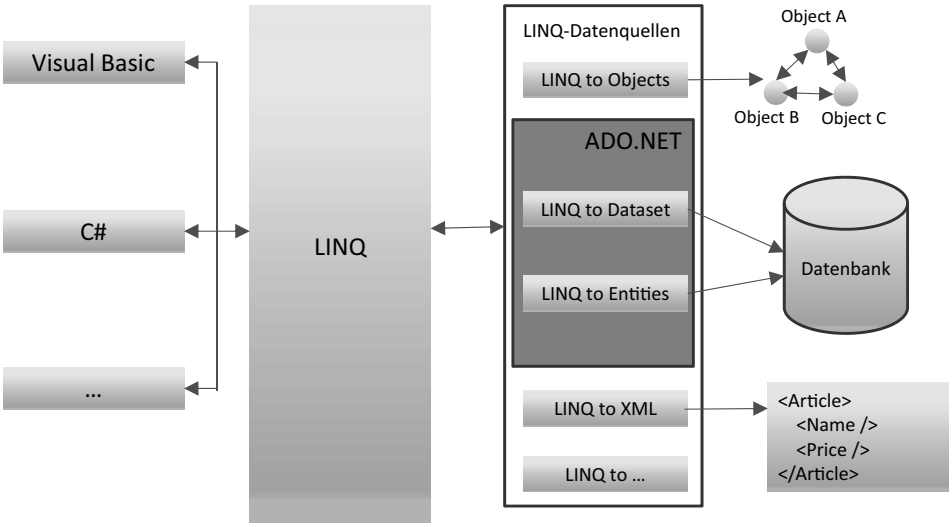

Bild 7.1 LINQ-Architektur

LINQ-Implementierungen

LINQ eröffnet zahlreiche Varianten für den Zugriff auf verschiedenste Arten von Daten. Diese sind in den verschiedenen LINQ-Implementationen (auch als „LINQ Flavors", d. h. „Geschmacksrichtungen", bezeichnet) enthalten. Folgende LINQ-Provider wurden bereits als Bestandteile des .NET-Frameworks bereitgestellt (siehe Bild 7.1):

- LINQ to Objects (arbeitet mit Collections, die *IEnumerable<T>* implementieren),
- LINQ to XML (Zugriff auf XML-Strukturen),
- LINQ to DataSet (arbeitet auf Basis von DataSets) und
- LINQ to Entities (verwendet das Entity Framework (Core) als ORM). Entities sind vom Typ *IQueryable<T>* anstatt von *IEnumerable<T>* wie bei LINQ to Objects.

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 kombiniert werden können.

HINWEIS: Wir werden uns in diesem Kapitel hauptsächlich auf **LINQ to Objects** beschränken, da dieser Provider die grundlegende Technologie bereitstellt und die übrigen Flavours mehr eine Angelegenheit der Datenbankliteratur sind.

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

Die Einführung von LINQ machte mehrere Ergänzungen bei den .NET-Programmiersprachen erforderlich, von denen einige bereits in diesem Kapitel (Lambda-Ausdrücke) bzw. im

einführenden Sprachkapitel 3 (Typinferenz) bzw. im OOP-Kapitel 4 (Objektinitialisierer) besprochen wurden. Auf zwei weitere Sprachfeatures (anonyme Typen und Erweiterungsmethoden), ohne die LINQ nicht möglich wäre, wollen wir im Folgenden eingehen.

7.1.2 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.

Die Deklaration anonymer Typen erfolgt mittels Typinferenz (Schlüsselwort *var*, siehe Abschnitt 3.2.10) und eines anonymen Objektinitialisierers. Das heißt, man lässt beim Initialisieren einfach den Klassennamen weg und schreibt stattdessen *var*. Der Compiler erzeugt die anonyme Klasse anhand der Eigenschaften im Objektinitialisierer und anhand des jeweiligen Typs der zugewiesenen Werte.

HINWEIS: In den meisten Beispielen, die Sie im Internet finden, wird fast nur noch *var* anstatt des korrekten Typnamens geschrieben, also auch bei einfachen Wertetypen. Dafür wurde das Schlüsselwort eigentlich nicht eingeführt, aber die Bequemlichkeit hat in den letzten Jahren wohl überhandgenommen. Bei sämtlichen Beispielen bis hierher haben wir auf *var* verzichtet, weil zumindest der Autor dieses Kapitels kein großer Fan davon ist. Ab jetzt werden wir aber *var* dort verwenden, wo wir es ohnehin brauchen, und auch an den Stellen, wo eindeutig erkennbar ist, für welchen Datentyp *var* steht.

Beispiel 7.1: Eine Objektvariable *person* wird aus einer anonymen Klasse instanziiert.

C#

```
var person = new { Vorname = "Jürgen", Nachname = "Kotz", Alter = 56};
```

Der Compiler generiert hierfür intern (in MSIL-Code) die folgende Klasse:

```
internal class ??????
{
    public string Vorname { get; set; }
    public string Nachname { get: set; }
    public int Alter { get; set; }
 }
```

HINWEIS: Da der Typ der Eigenschaften aus der jeweiligen Klasse bzw. Struktur des im Objektinitialisierer zugewiesenen Werts abgeleitet wird, darf man hier nicht *null* zuweisen, denn der Compiler kann in diesem Fall den Datentyp der Eigenschaft nicht bestimmen.

Beispiel 7.2: Fehlerhafter Code, der Compiler kann den Typ der Eigenschaft *Alter* nicht ableiten.

C#

```
var person = new { Vorname = "Jürgen" , Nachname = "Kotz", Alter = null};
// Fehler!!!
```

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

Beispiel 7.3: Da Name, Typ und Reihenfolge der Eigenschaften im Objektinitialisierer bei *person* (siehe oben) und *kunde* identisch sind, ist eine direkte Zuweisung möglich.

C#

```
var kunde = new { Vorname = "Lennard", Nachname = "Kotz", Alter = 18};
person = kunde;
```

HINWEIS: Anonyme Typen dürfen nur lokal verwendet werden. Sie sind auch als Übergabeparameter nicht erlaubt.

7.1.3 Erweiterungsmethoden

Normalerweise erlaubt eine objektorientierte Programmiersprache das Erweitern von Klassen durch Vererbung. Bereits C# 3.0 führte aber eine neue Syntax ein, die das direkte Hinzufügen neuer Methoden zu einer vorhandenen Klasse erlaubt. Diese sogenannten *Erweiterungsmethoden* werden als statische Methoden in einer neuen statischen Klasse implementiert und können dann wie eine normale Methode (d. h. Instanzmethode) des erweiterten Datentyps aufgerufen werden. Um eine Methode als Erweiterungsmethode zu deklarieren, wird vor dem ersten Parameter das Schlüsselwort *this* angegeben. Der Argumenttyp des ersten Parameters bezieht sich auf die zu erweiternde Klasse bzw. Struktur. Wird die Erweiterungsmethode dann aufgerufen, übergibt der Compiler die Instanz des erweiterten Typs als erstes Argument an die Methode.

HINWEIS: Erweiterungsmethoden können auch für Interfaces definiert werden.

Beispiel 7.4: Zwei Erweiterungsmethoden (*Multiply* und *Abs*) für die Basisklasse *System.Int32*

C#

```
public static class IntExtension
{
    // 1. Erweiterungsmethode
    public static int Multiply(this int instanz, int faktor)
    {
```

```
        return instanz * faktor;
    }
    // 2. Erweiterungsmethode
    public static int Abs(this int instanz)
    {
       if (instanz < 0)
       {
           return -1 * instanz;
       }
       return instanz;
    }
}
```

Der Test:

```
int zahl = -95;
textBox1.Text = zahl.Multiply(7).ToString();     //  -665
textBox2.Text = zahl.Abs().ToString();           //    95
```

In diesem Beispiel kann man nun die Erweiterungsmethoden *Multiply* 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.

HINWEIS: Befindet sich die statische Klasse, in der die Extensionmethoden definiert sind, in einem anderen Namespace als die der aufrufenden Klasse, so muss unbedingt der Namespace der statischen Klasse mittels *using* bekannt gemacht werden. IntelliSense schlägt Ihnen die Extensionmethoden erst dann vor, wenn auch der Namespace bekannt ist.

7.2 Abfragen mit LINQ to Objects

LINQ stellt bekanntlich grundlegende Abfragefunktionen in einer SQL-ähnlichen Syntax bereit. Dazu gehören zunächst als wichtigster Standard das Angeben einer Quelle (*from*), das Festlegen der zurückzugebenden Daten (*select*), das Filtern (*where*) und das Sortieren (*orderby*). Hinzu kommen eine Fülle weiterer Operatoren, wie z. B. für das Gruppieren, Verknüpfen und Sammeln von Datensätzen, auf die wir aber erst später eingehen wollen.

7.2.1 Grundlegendes zur LINQ-Syntax

Die LINQ-Abfrageoperatoren sind als Erweiterungsmethoden (siehe Abschnitt 7.1.3) definiert und in der Regel auf beliebige Objekte, die *IEnumerable<T>* implementieren, anwendbar.

Beispiel 7.5: Gegeben sei die folgende Auflistung.

C#

```
string[] monate = { "Januar", "Februar", "März", "April", "Mai", "Juni",
        "Juli", "August", "September", "Oktober", "November", "Dezember" };
```

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

```
var expr = from s in monate
           where s.Length == 6
           orderby s
           select s.ToUpper();
```

Die Ergebnisanzeige:

```
foreach (string item in expr)
{
    listBox1.Items.Add(item);
}
```

Das Resultat in einem Listenfeld:

```
AUGUST
JANUAR
```

Obiges Beispiel demonstriert das allgemeine Format einer LINQ-Abfrage:

Syntax:

```
from … < where … orderby … > 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 davon irritiert sein, dass eine LINQ-Abfrage mit *from* und nicht mit *select* beginnt. Der Grund hierfür ist, 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. Vielleicht haben Sie diese Unterstützung im SQL Server Management Studio schon mal vermisst.

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

```
var expr = from s in monate …
```

ist für den Compiler identisch mit

```
IEnumerable<string> expr = from s in monate …
```

7.2.2 Zwei alternative Schreibweisen von LINQ-Abfragen

Grundsätzlich sind für LINQ-Abfragen zwei gleichberechtigte Schreibweisen möglich:

- Query-Expression-Syntax[1] (Abfrage-Syntax)
- Extension-Method-Syntax (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 7.6: Die LINQ-Abfrage des obigen Beispiels in Extension-Method-Syntax

C#

```
var expr = monate
                .Where(s => s.Length == 6)
                .OrderBy(s => s)
                .Select(s => s.ToUpper());
```

Oder kompakt in einer Zeile:

```
var expr = monate.Where(s => s.Length == 6).OrderBy(s => s).
                  Select(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 7.7: Obiges Beispiel in gemischter Syntax

C#

```
var expr = (from s in monate where s.Length == 6  select s.ToUpper()).
           OrderBy(s => s);
```

Hier wurde ein Abfrageausdruck in runde Klammern eingeschlossen, gefolgt von der Erweiterungsmethode *OrderBy*. So lange, 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. Der Compiler kompiliert alle Queries in die Extension-Method-Syntax, die der objektorientierte Programmierung näher steht. 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.

[1] Die *Extension-Method-Syntax* wird auch als *Dot-Notation-Syntax* bezeichnet und entspricht der Objektorientierung etwas mehr.

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*, *orderby*, *groupby* und *join*. Wie bereits erwähnt, lassen sich diese Klauseln in die gleichwertigen Operatoraufrufe 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.

7.2.3 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<T>* implementieren.

HINWEIS: Kommen bei der Extension-Method-Syntax (Erweiterungsmethoden-Syntax) Abfrageoperatoren bzw. -methoden zur Anwendung, so sollten wir bei der Query-Expression-Syntax (Abfragesyntax) präziser von Abfrageklauseln 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*, *Chunk*
Mengenoperatoren (Sets)	*Distinct*, *Union*, *Intersect*, *Except*
Elementoperatoren (Elements)	*First*, *FirstOrDefault*, *ElementAt*
Aggregatoperatoren (Aggregation)	*Count*, *Sum*, *Min*, *Max*, *Average*
Konvertierungsoperatoren (Conversion)	*ToArray*, *ToList*, *ToDictionary*
Typumwandlungsoperatoren (Casting)	*OfType<T>*

Bild 7.2 illustriert an einem Beispiel, wie einige der bereits im Vorgängerabschnitt diskutierten neuen Sprachfeatures in LINQ-Konstrukten zur Anwendung kommen und wie die Abfragesyntax vom Compiler in die äquivalente Erweiterungsmethoden-Syntax umgesetzt wird.

Vergleich zwischen Abfragesyntax (oben) und Erweiterungsmethoden-Syntax (unten):

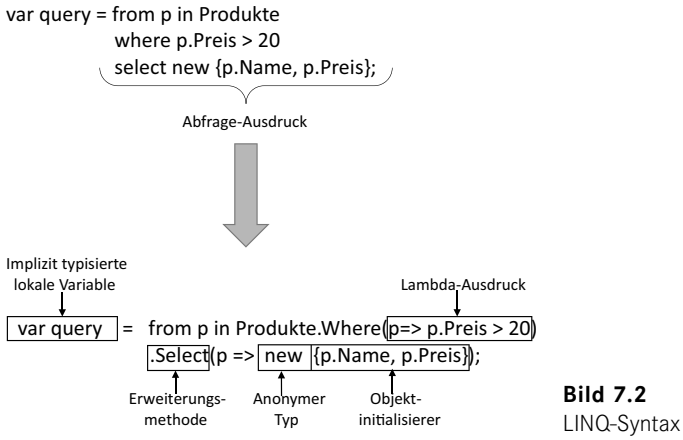

Bild 7.2 LINQ-Syntax

7.3 LINQ-Abfragen im Detail

Das Ziel der folgenden Beispiele ist nicht die vollständige Erläuterung aller in der obigen 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 Diagramm zeigt.

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

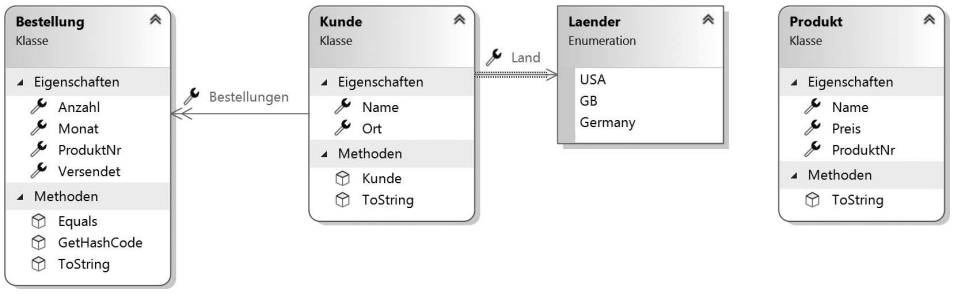

Bild 7.3 Datenmodell für unsere Beispiele

7.3.1 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<T>* implementiert.

Beispiel 7.8: Namen aller Produkte ausgegeben (Extension-Method-Syntax)

C#

```
var allProdukte = Produkte.Select(p => p.Name);
```

Alternativ die Query-Expression-Syntax:

```
var allProdukte = from p in Produkte select p.Name;
```

Die Ausgabe der Ergebnisliste:

```
listBox1.DataSource = allProdukte.ToList();
```

Ergebnis

Der Inhalt des Listenfelds sollte dann etwa folgenden Anblick bieten:

```
Marmelade
Quark
Mohrrüben
...
```

Beispiel 7.9: Das Abfrageergebnis wird auf einen anonymen Typ projiziert, der als Tupel definiert ist.

C#

```
var expr = Kunden.Select(k => new {k.Name, k.Ort});
```

Alternativ die Query-Expression-Syntax:

```
var expr = from k in Kunden
           select new { K.Name, k.Ort };
listBox1.DataSource = expr.ToList();
```

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<Bestellung>* 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 7.10: Die Bestellungen aller Kunden aus Deutschland sollen ermittelt werden.

C#

```
var bestellungen = Kunden
                .Where(k => k.Land == Länder.Germany)
                .SelectMany(k => k.Bestellungen);
```

Alternativ der Abfrageausdruck in Query-Expression-Syntax:

```
var bestellungen =
                from k in Kunden
                where k.Land == Länder.Germany
                from b in k.Bestellungen
                select b;
```

Das Auslesen des Ergebnisses der Abfrage:

```
listBox1.DataSource = bestellungen.ToList();
```

Die Ausgabe (Voraussetzung ist eine entsprechende Überschreibung der *ToString()*-Methode der Klasse *Bestellung*):

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

7.3.2 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 7.11: Die Kunden an den Positionen 2 und 3 der Kundenliste sollen angezeigt werden.

C#

```
var expr =  Kunden
            .Where((k, index) => (index >= 2) && (index < 4))
            .Select(k => k.Name);
listBox1.DataSource = expr.ToList();
```

Ergebnis

```
Holger
Fernando
```

7.3.3 Die Sortierungsoperatoren OrderBy und ThenBy

Diese Operatoren bewirken ein Sortieren der Elemente innerhalb der Ergebnismenge.

OrderBy/OrderByDescending

Dieses nette Pärchen ermöglicht das Sortieren in auf- bzw. absteigender Reihenfolge.

Beispiel 7.12: Alle Produkte mit einem Preis kleiner gleich 20 sollen ermittelt und nach dem Preis sortiert ausgegeben werden (das teuerste zuerst).

C#

```
var prod = Produkte
          .Where(p => p.Preis <= 20)
          .OrderByDescending(p => p.Preis)
          .Select(p => new { p.Name, p.Preis });
```

Oder alternativ als Abfrageausdruck:

```
var prod = from p in Produkte
           where p.Preis <= 20
           orderby p.Preis descending
           select new { p.Name, p.Preis };
```

Die Ausgabe:

```
listBox1.DataSource = prod.ToList();
```

Ergebnis

```
{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<T>*, sondern nur auf den Typ *IOrderedSequence<T>* anwendbar sind, können diese Operatoren nur im Anschluss an *OrderBy/OrderByDescending* eingesetzt werden.

Beispiel 7.13: Alle Kunden sollen zunächst nach ihrem Land und dann nach ihren Namen sortiert werden.

C#

```
var knd = Kunden
          .OrderBy(k => k.Land)
          .ThenBy(k => k.Name)
          .Select(k => new { k.Land, k.Name});
```

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
          orderby k.Land, k.Name
          select new { k.Land, k.Name };
```

Die Ausgabe ...

```
listBox1.DataSource = knd.ToList();
```

Ergebnis

... 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 Reihenfolge der Elemente im Abfrageergebnis umzukehren. Der Operator ist als Abfrageausdruck nicht verfügbar.

Beispiel 7.14: Das Vorgängerbeispiel mit umgekehrter Reihenfolge der Ergebniselemente

C#

```
var knd = Kunden.OrderBy(k => k.Land).ThenBy(k => k.Name).
                 Select(k => new { k.Land, k.Name }).Reverse();
```

Ergebnis

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

7.3.4 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 7.15: Alle Kunden nach Ländern gruppieren

C#

```
var knd = Kunden
          .GroupBy(k => k.Land);
```

Der alternative Abfrageausdruck:

```
var knd = from k in Kunden
          group k by k.Land;
```

Durchlaufen der Ergebnismenge:

```
            foreach (IGrouping<Laender, Kunde> kdGroup in knd)
            {
                listBox1.Items.Add(kdGroup.Key);
```

```
                foreach (var kd in kdGroup)
                {
                    listBox1.Items.Add($"  {kd}");
                }
            }
```

Ergebnis

Der Gruppenschlüssel (*kdGroup.Key*) ist hier das Land. Die Standardausgabe der Gruppenelemente erfolgt entsprechend der überschriebenen *ToString()*-Methode der Klasse *Kunden* (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<IGrouping<K, T>>* liefern. Die generische Schnittstelle *IGrouping<K, T>* definiert einen spezifischen Schlüssel vom Typ *K* für die Gruppenelemente (Typ *T*).

Der Typ der äußeren Schleifenvariablen *IGrouping<Laender, Kunde>* kann auch implizit deklariert werden, sodass sich der Schleifenkopf im obigen Beispiel wie folgt vereinfachen lässt:

```
            foreach(var kdGroup in knd)
```

Wenn nicht der standardmäßige, sondern ein benutzerdefinierter Elemente-Selektor zum Einsatz kommen soll, muss eine weitere Überladung von *GroupBy* verwendet werden (siehe folgendes Beispiel).

Beispiel 7.16: Das gleiche Problem wie im Vorgängerbeispiel wird gelöst, als Gruppenelemente werden allerdings nur die Namen der Kunden ausgegeben.

C#

```
var knd = Kunden
          .GroupBy(k => k.Land, k => k.Name);
          foreach (var kdGroup in knd)
          {
               listBox1.Items.Add(kdGroup.Key);
               foreach (var kd in kdGroup)
               {
                   listBox1.Items.Add($"  {kd}");
               }
          }
```

Ergebnis

```
Germany
   Walter
   Thomas
```

```
USA
   Holger
   Fernando
GB
   Alice
```

Beispiel 7.17: Alle Produkte werden nach ihren Anfangsbuchstaben gruppiert.

C#

```
var prodGroups = Produkte
                  .GroupBy(p => p.Name[0], p => p.Name);
                  foreach (var pGroup in prodGroups)
// var ersetzt IGrouping<char, string>
                  {
                      listBox1.Items.Add(pGroup.Key);
                      foreach (var p in pGroup)
                      {
                          listBox1.Items.Add($"  {p}");
                      }
                  }
```

Ergebnis

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

C#

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

```
var prodGroups = from p in Produkte
                  group p by p.Name[0] into g
                  select new { firstLetter = g.Key, prods = g };
                  foreach (var g in prodGroups)
                  {
                     listBox1.Items.Add(g.firstLetter);
                     foreach (var p in g.prods)
                     {
                         listBox1.Items.Add(p.Name);
                     }
                  }
```

7.3.5 Verknüpfen mit Join

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

Beispiel 7.18: Die Bestellungen aller Kunden werden aufgelistet.

C#

```
var bestprod = Kunden.SelectMany(k => k.Bestellungen)
                     .Join(Produkte, b => b.ProduktNr, p => p.ProduktNr,
                      (b, p) => new {
                      b.Monat, p.ProduktNr, p.Name, p.Preis, b.versendet });
```

Alternativ die Notation in Abfragesyntax:

```
var bestprod = from k in Kunden
               from b in k.Bestellungen
               join p in Produkte on b.ProduktNr equals p.ProduktNr
               select new { 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. Ein Vergleich mit dem Vergleichsoperator == ist ebenfalls nicht zulässig.

Die Anzeigeroutine:

```
listBox1.DataSource = bestprod.ToList();
```

Ergebnis

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}
...
```

7.3.6 Aggregat-Operatoren

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

C#

```
var kdn = Kunden.Select(k => new {
                    k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count() });
```

Oder das Gleiche in Abfragesyntax:

```
 var kdn = from k in Kunden
           select new { k.Name, k.Ort, AnzahlBest = k.Bestellungen.Count()};
```

Wir zeigen die Ergebnismenge in der ListBox an:

```
listBox1.DataSource = kdn.ToList();
```

Ergebnis

```
{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 7.20: Die Summe aller Preise der Produktliste

C#

```
var total = Produkte.Sum(p => p.Preis);
```

Die alternative Abfragesyntax (eigentlich gemischte Syntax):

```
var 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 7.21: Die Gesamtsumme aller aufgegebenen Bestellungen wird ermittelt.

C#

```
var 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 { k.Name, BestellBetrag = b.Anzahl * p.Preis }
                on k.Name equals b.Name into KundenMitBest
                select new { k.Name,
                    TotalBetrag = KundenMitBest.Sum(b => b.BestellBetrag)};
listBox1.DataSource = expr.ToList();
```

Ergebnis

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

Das schaut jetzt schon nicht mehr ganz so trivial aus. Etwas übersichtlicher ist die Erweiterungsmethodensyntax:

```
var expr = Kunden.Select(k => new { k.Name,
                         TotalerBetrag = k.Bestellungen.Sum(b => b.Anzahl *
                         Produkte.First(p => p.ProduktNr == b.ProduktNr).Preis) });
```

Dadurch, dass bestimmte Operationen in der Abfragesyntax nicht möglich sind und wir deswegen sowieso die Erweiterungsmethodensyntax bzw. einen Mix verwenden müssen, ist der Favorit der Autoren eindeutig die Erweiterungsmethodensyntax. Deswegen werden wir in den folgenden Kapiteln die Abfragesyntax nicht mehr darstellen. Dem objektorientierten Programmierer, und das wollen Sie doch werden, dürfte die Punkt-Syntax sowieso besser gefallen.

7.3.7 Verzögertes Ausführen von LINQ-Abfragen

Normalerweise werden LINQ-Ausdrücke nicht bereits bei ihrer Definition, sondern erst bei Verwendung der Ergebnismenge ausgeführt (*deferred Execution*). 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 7.22: Alle Produkte, die mit dem Buchstaben „M“ beginnen, sollen ermittelt werden.

C#

```
var prods = Produkte.Where(p => p.Name[0] == 'M').Select(p => p.Name);
```

Oder alternativ in Abfragesyntax:

```
var prods = from p in Produkte where p.Name[0] =='M' select p.Name;
```

Die Ergebnismenge wird das erste Mal durchlaufen und angezeigt:

```
foreach (var prod in prods)
{
     listBox1.Items.Add(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:

```
foreach (var prod in prods)
{
     listBox1.Items.Add(prod);
}
```

Ergebnis

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 ausgeführt wird, wenn wir (wie hier in der *foreach*-Schleife) auf das Abfrageergebnis (*prods*) zugreifen.

Abfragen dieser Art bezeichnet man deshalb auch als „verzögerte Abfragen“.[2] Mitunter aber ist dieses Verhalten nicht erwünscht, d. h. man möchte das Abfrageergebnis nicht verzögert, sondern sofort nach Definition der Abfrage zur Verfügung haben. Abhilfe schafft hier die im nächsten Abschnitt beschriebene Anwendung von Konvertierungsmethoden.

HINWEIS: Wenn Sie mittels des LINQ-Operators *First* oder *FirstOrDefault* nur den ersten Satz einer Ergebnismenge lesen wollen (respektive *Last* oder *LastOrDefault*), dann wird die Abfrage auch sofort ausgeführt.

7.3.8 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.

[2] *deferred query execution*

Beispiel 7.23: Das Vorgängerbeispiel wird wiederholt, diesmal aber wird das Abfrageergebnis in einer generischen List zwischengespeichert.

C#

```
var mProds = (Produkte.Where(p => p.Name[0] == 'M').
              Select(p => p.Name)).ToList();
```

bzw.:

```
var mProds = (from p in Produkte where p.Name[0] == 'M' select p.Name).ToList();
```

Ergebnis

Die Änderung der Quellfolge bleibt ohne Konsequenz für das Abfrageergebnis:

```
Produkte[0].Name = "Milch";
...
```

Ausgabe:

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

7.3.9 Abfragen mit PLINQ

PLINQ ist eine parallele Implementierung von LINQ to Objects und kombiniert die Einfachheit und Lesbarkeit der LINQ-Syntax mit der Leistungsfähigkeit der parallelen Programmierung. PLINQ besitzt das komplette Angebot an Standard-Abfrageoperatoren und hat zusätzliche Operatoren für parallele Operationen.

Als Reaktion auf die zunehmende Verfügbarkeit von Mehrprozessorplattformen bietet PLINQ eine einfache Möglichkeit, die Vorteile paralleler Hardware einschließlich herkömmlicher Mehrprozessorcomputer und der neueren Generation von Mehrkernprozessoren zu nutzen.

HINWEIS: In vielen Szenarien kann PLINQ signifikant die Geschwindigkeit von LINQ-to-Objects-Abfragen steigern, da es alle verfügbaren Prozessoren des Computers 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 weiterverwenden, 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. *AsParallel* kann auf jeder Datenmenge ausgeführt werden, die *IEnumerable<T>* implementiert.

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

Beispiel 7.24: Eine einfache LINQ-Abfrage über eine Liste von Integer-Zahlen

C#

```
List<int> zahlen = new List<int>() { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Oder auch:

```
IEnumerable<int> zahlen = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var q = from x in zahlen
        where x > 3
        orderby x descending
        select x;
```

Erst beim Iterieren über die Liste wird die Abfrage ausgeführt:

```
foreach (var z in q)
{
    listBox1.Items.Add(z.ToString());
    // 9, 8, 7, 6, 5, 4
}
```

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

Beispiel 7.25: Die Abfrage im obigen Beispiel mit PLINQ

C#

```
...
var q = from x in zahlen.AsParallel()
        where x > 3
        orderby x descending
        select x;
...
```

Die Abfragen in den obigen Beispielen wurden in Query-Expression-Syntax geschrieben. Alternativ kann man natürlich auch die Extension-Method-Syntax[3] verwenden.

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

Beispiel 7.26: Beide obigen Abfragen in Erweiterungsmethoden-Syntax

C#

Einfache LINQ-Version:

```
…
var q = zahlen
        .Where(x => x > 3)
        .OrderByDescending(x => x)
        .Select(x => x);
...
```

PLINQ-Version:

```
…
 var q = zahlen.AsParallel()
        .Where(x => x > 3)
        .OrderByDescending(x => x)
        .Select(x => x);
...
```

Nach dem Aufruf der *AsParallel*-Methode führt PLINQ transparent die Erweiterungsmethoden (*Where*, *OrderBy*, *Select*, …) auf allen verfügbaren Prozessoren aus. Genauso wie LINQ realisiert auch PLINQ eine verzögerte Ausführung von Abfragen, d. h. erst beim Durchlaufen der *foreach*-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.

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, haben 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.[4]

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

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

Beispiel 6.27: Verwendung von *AsOrdered*

C#

```
List<int> zahlen = new List<int>()
       { 7, 4, 2, 3, 1, 6, 11, 5, 10, 8, 9, 13, 12 };
var q = (from x in zahlen.AsParallel().AsOrdered()
where x > 3
select x).Take(5);
```

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 kann auch

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

ausgegeben werden.

Wie sich Sortierfolgen auf bestimmte Operatoren auswirken, beschreibt im Detail diese Webseite: *http://msdn.microsoft.com/de-de/library/dd460677(v=vs.110).aspx*.

HINWEIS: Grundsätzlich gilt jedoch: Vermeiden Sie im Zusammenhang mit PLINQ die Anwendung von Sortieroperationen. Diese machen die Vorteile von PLINQ durch erhöhten Verwaltungsaufwand meist wieder zunichte. Für Aufgaben, bei denen die Sortierung egal ist, erweist sich PLINQ jedoch als eine performancegewinnende Technologie.

7.4 Praxisbeispiele

7.4.1 Die Syntax von LINQ-Abfragen verstehen

In diesem Praxisbeispiel lernen Sie den prinzipiellen Aufbau von LINQ-Abfragen kennen. Im Zusammenhang damit kommen Sprachfeatures wie Typinferenz, Lambda-Ausdrücke und Erweiterungsmethoden zum Einsatz.

Prinzipiell gibt es zwei verschiedene Syntaxformen für LINQ-Abfragen:

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

Im Folgenden werden wir beide Syntaxformen demonstrieren, um den Inhalt eines Integer-Arrays zu verarbeiten. Außerdem wird eine Mischform vorgeführt.

Oberfläche

Auf dem Startformular *Form1* befinden sich eine *ListBox* und drei *Button*s (siehe Bild 7.4).