

3 Operatoren

Die vielfältigen Funktionen, die LINQ bereitstellt, werden Operatoren genannt. Diese Operatoren verändern oder erzeugen eine Sequenz und leiten sie oftmals an den nächsten Operator weiter, der möglicherweise wiederum dasselbe tut. Die Operatoren zu kennen, ist daher eine wichtige Voraussetzung, um erfolgreich mit LINQ zu programmieren.

Dieses Kapitel stellt alle Operatoren mit ihren entsprechenden Überladungen vor. Für eine bessere Übersicht sind diese Operatoren in Gruppen eingeteilt. Diese Gruppeneinteilung soll helfen, schnell den gesuchten Operator für eine bestimmte Aufgabe zu finden.

Gruppe	Funktionen
Projektionsoperatoren	Projektion der Elemente in die abschließende Form.
Filteroperatoren	Beliebiges Filtern mit jeglichem C#-Ausdruck, der <code>true</code> oder <code>false</code> liefern kann.
Sortierungsoperatoren	Auf- oder absteigende, mehrfache Sortierung.
Gruppierungsoperatoren	Gruppierung nach gleichen Schlüsselwerten.
Join-Operatoren	Beziehungen zwischen Sequenzen über Schlüsselwerte.
Mengenoperatoren	Operatoren zum Erstellen von Schnitt- und Vereinigungsmengen etc.
Aggregationsoperatoren	Funktionen zum Berechnen von Summen, Minimum und Maximum etc.
Generierungsoperatoren	Operatoren zum Erzeugen von Sequenzen.
Quantifizierungsoperatoren	Bewertung von Sequenzen. Sind z.B. bestimmte Bedingungen erfüllt oder bestimmte Elemente vorhanden?
Aufteilungsoperatoren	Funktionalitäten zum Aufteilen von Sequenzen, also z.B. die ersten oder letzten <code>n</code> Elemente.
Elementoperatoren	Zugriff auf einzelne Elemente einer Sequenz ohne <code>foreach</code> -Schleife (Iterator).
Konvertierungsoperatoren	Operatoren zum Konvertieren von Sequenzen in Auflistungen (oder umgekehrt).
Sonstige Operatoren	Sonstige Operatoren, die in keine der anderen Gruppen passen.

Tabelle 3.1: Die Gruppen, in die die LINQ-Operatoren eingeteilt werden

Für alle Operatoren werden deren Signaturen abgedruckt. So ist leicht zu erkennen, ob und wie ein Operator überladen ist. Wie aber sind solche Signaturen zu lesen?

Neben den Aspekten der generischen Programmierung ist zu berücksichtigen, dass alle Operatoren Erweiterungsmethoden sind und somit der erste Parameter nach dem `this`-

Schlüsselwort die Schnittstelle angibt, die erweitert wird. In den meisten Fällen ist dies die `IEnumerable<>`-Schnittstelle.

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source, // Diese Schnittstelle wird erweitert
    Func<TSource, TResult> selector); // Func<> ist ein Delegat
```

`Func<>` in der Signatur bezeichnet ein generisches Delegat, dessen letzter Parameter den Typ der Rückgabe angibt. Diese Delegaten können mit gewöhnlichen oder anonymen Methoden und einfachen Lambda-Ausdrücken verwendet werden. Da Lambda-Ausdrücke wohl die häufigste Variante bei LINQ sind, sehen Sie im Folgenden den Ausdruck, der zu der Signatur oben passt:

```
TSource => TResult oder (TSource) => TResult
```

Hat `Func<TResult>` nur einen Parameter, so ist dies ebenfalls wieder der Rückgabewert (da dieser ja auch wieder den letzten Parameter darstellt). Der Lambda-Ausdruck sieht in diesem Fall so aus:

```
() => TResult
```

Das leere Klammernpaar ist hier unabdingbar, da der Compiler sonst einen Fehler erzeugt.

Als letzte Möglichkeit können mehr als drei Parameter vorliegen. In diesem Fall werden die Parameter eines Lambda-Ausdrucks in runde Klammern gefasst. Für `Func<TSource1, TSource2, TSource3, TResult>` sieht der Ausdruck entsprechend so aus:

```
(TSource1, TSource2, TSource3) => TResult
```

Da zu jeder soliden Programmierung auch die Berücksichtigung von möglichen Fehlern in Form einer Ausnahmebehandlung gehört, finden Sie im Folgenden eine Auflistung der möglichen Ausnahmen, die der Operator auslösen kann, zusammen mit einer kurzen Erklärung in welcher Situation er dies tut.

Zur Abrundung werden mittels kleiner Beispiele die letzten Unklarheiten über die sinnvolle Anwendung beseitigt. Dabei werden die wichtigen Stellen fett hervorgehoben. Damit Sie beim Lesen das gute Gefühl bekommen, verstanden zu haben, was der Operator tut, haben die meisten Beispiele eine Ausgabe, die sich im Anschluss des Codes befindet (aus Platzgründen oft in leicht gekürztem Umfang).

3.1 Projektionsoperatoren

Dieser Abschnitt beschreibt die so genannten Projektionsoperatoren, deren Aufgabe es ist, die gewünschten Inhalte aus einer Quellsequenz in eine Ergebnissequenz zu übertragen. Dies geschieht nach Filterung und Sortierung durch andere Operatoren – Projektionsoperatoren werden daher meist abschließend ausgeführt, um zu bestimmen, wie das

Endergebnis aussehen soll. Ihre Ausgabe kann jedoch auch wiederum die Grundlage (Quelle) einer weiteren LINQ-Abfrage darstellen.

Für beide Operatoren gilt, dass sie erst direkt beim Durchlaufen einer foreach-Schleife oder beim Aufruf der GetEnumerator()-Methode ausgeführt werden und immer nur jeweils ein Element nach dem anderen liefern. Sie folgen damit den Regeln der verzögerten Ausführung und ihr Ergebnis kann sich zwischen zwei Aufrufen verändern, wenn die Quellsequenz verändert wurde.

Hinweis

Wird eine stabile Auflistung benötigt, die sich nicht bei Veränderungen an der Quellsequenz ändert, so kann dies mit einem Konvertierungsoperator wie ToArray, ToList, ToDictionary oder ToLookup realisiert werden. Diese erzeugen eine Auflistung im klassischen Sinne. Für Details siehe Abschnitt 3.12 *Konvertierungsoperatoren*.

3.1.1 Select

Der Select-Operator projiziert das Ergebnis auf eine Sequenz, die die IEnumerable<T>-Schnittstelle implementiert. Dabei wird eine flache Liste erstellt, die z.B. mit einer foreach-Schleife durchlaufen werden kann. Daher ist der Select-Operator oftmals der letzte in der Reihe, um die gefilterten und aufbereiteten Elemente in die finale Ergebnissequenz zu projizieren. Durch eine zweite Überladung können die Elemente beginnend mit null durchnummeriert werden oder der laufende Index kann für eine Berechnung verwendet werden.

Dieser Operator steht sowohl in der LINQ-Syntax (für die erste Überladung) als auch als Methode zur Verfügung.

Überladungen

Hier die Signaturen der beiden Überladungen des Select-Operators.

Überladung 1:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector);
```

Überladung 2:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, TResult> selector);
```

Ausnahmen

Es wird eine Ausnahme vom Typ ArgumentNullException ausgelöst, wenn **source** oder **selector** lediglich null sind.

Beispiele

Überladung 1:

Das folgende Beispiel liefert die Namen und die physikalischen Pfade aller Systemordner über die `System.Environment.GetFolderPath()`-Methode.

```
// .NET-Typ der Aufzählung speichern
Type enumType = typeof(Environment.SpecialFolder);

// LINQ-Syntax
var folders1 = from sf in Enum.GetNames(enumType)
    select new {
        Enum = sf,
        Pfad = Environment.GetFolderPath(
            (Environment.SpecialFolder)Enum.Parse(enumType,sf))
    };

// Methode
var folders2 = Enum.GetNames(enumType).Select(sf =>
    new {
        Enum = sf,
        Pfad = GetFolderPath((SpecialFolder)Enum.Parse(enumType, sf))
    });

// Ausgabe
foreach (var folder in folders1)
    Debug.WriteLine(folder);
```

Die Ausgabe ist in beiden Fällen gleich und sieht exemplarisch so aus.

```
{ Enum = Desktop, Pfad = C:\Users\tkansy\Desktop }
{ Enum = Personal, Pfad = C:\Users\tkansy\Documents }
{ Enum = MyDocuments, Pfad = C:\Users\tkansy\Documents }
```

Überladung 2:

In der zweiten Überladung kann zusätzlich auf den laufenden Index für die Erstellung der Zielelemente zugegriffen werden. Dies wird ausgenutzt, um über einen Lambda-Ausdruck die Namen der Sequenz durchzunummerieren.

```
// Array mit Namen erstellen
String[] Namen = { "Julia", "Ursula", "Christiane", "Christine" };
var namen = Namen.Select(
    (name, index) => new { Vorname = String.Format("#{0}, {1}", index, name) }
);
```

```
// Ausgabe
foreach (var name in namen)
    Debug.WriteLine(name);
```

Die Ausgabe besteht also aus nummerierten Namen:

```
{ Vorname = #Julia, 0 }
{ Vorname = #Ursula, 1 }
{ Vorname = #Christiane, 2 }
{ Vorname = #Christine, 3 }
```

3.1.2 SelectMany

Während der Select-Operator im Kern aus einer Liste eine andere erzeugt, geht der SelectMany-Operator einen Schritt weiter. Er arbeitet auf Sequenzen, deren einzelne Elemente wiederum ebenfalls Sequenzen darstellen. Er durchläuft diese verschachtelten Sequenzen und erzeugt dabei eine flache Sequenz, die der Reihe nach die Elemente der einzelnen Sequenzen enthält.

Dieser Operator steht nur als Methode zur Verfügung.

Überladungen

Für diesen Operator existieren vier Überladungen.

Überladung 1:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector);
```

Überladung 2:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TResult>> selector);
```

Überladung 3:

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector);
```

Überladung 4:

```
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, int, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector);
```

Ausnahmen

Es wird eine Ausnahme vom Typ `ArgumentNullException` ausgelöst, wenn **source**, **selector**, **collectionSelector** oder **resultSelector** lediglich null sind.

Beispiele

Lassen Sie uns die ersten drei Überladungen jeweils an einem kleinen Beispiel demonstrieren.

Überladung 1:

Das Beispiel für die einfachste Überladung dieses Operators erzeugt aus einer Liste aus Sätzen eine flache Liste mit Wörtern. Alternativ wird die Ausgabe mit zwei verschachtelten `foreach`-Schleifen dargestellt, die ohne den `SelectMany`-Operator notwendig sind.

```
// Eine Liste an fiesen Zungenbrechern
string[] Zungenbrecher = new string[] {
    "Fischers Fritze fischt frische Fische.",
    "Brautkleid bleibt Brautkleid und Blaukraut bleibt Blaukraut.",
    "Der Kaplan klebt klappbare Pappplakate";

// Dieses Array aus Zungenbrechern lässt sich leicht in eine Sequenz
// aus String Arrays aufteilen.
IEnumerable<string[]> Wortlisten = Zungenbrecher.Select(w => w.Split(' '));

// Damit werden zwei verschachtelte foreach-Schleifen benötigt,
// um alle Wörter auszugeben
foreach (string[] Woerter in Wortlisten)
    foreach (string Wort in Woerter)
        Debug.WriteLine(Wort);

// Einfacher geht es mit dem SelectMany-Operator,
// der eine flache Liste erzeugt und nur eine Schleife
// erforderlich macht.
IEnumerable<string> flacheWortliste = Wortlisten.SelectMany( c =>c );

// Alternativ können einzelne Elemente der endgültigen flachen Liste
// z.B. von Punkten am Ende bereinigt werden.
```

```
IEnumerable<string> flacheWortliste2 =  
    Wortlisten.SelectMany( c=>c.Select( d=>d.TrimEnd('.') ) );  
  
// Ausgabe  
foreach (string Wort in flacheWortliste)  
    Debug.WriteLine(Wort);
```

Die Ausgabe der einfachen Schleife besteht aus einer Abfolge von Zeichenketten, welche aus den Wörtern der Zungenbrecher besteht.

```
Fischers  
Fritze  
fischt  
frische  
Fische.
```

Überladung 2:

Im Beispiel für die zweite Überladung kann auf den laufenden Index für die Erstellung der Zielelemente zugegriffen werden. Da sich dieser auf die Quellsequenzen bezieht, wird ein weiterer Select-Operator für den Index des Wortes innerhalb eines Zungenbrechers verwendet.

```
// Eine Liste an diesen Zungenbrechern  
string[] Zungenbrecher = new string[] {  
    "Fischers Fritze fischt frische Fische.",  
    "Brautkleid bleibt Brautkleid und Blaukraut bleibt Blaukraut.",  
    "Der Kaplan klebt klappbare Pappplakate";  
  
// Dieses Array aus Zungenbrechern lässt sich leicht in eine Sequenz  
// aus String Arrays aufteilen.  
IEnumerable<string[]> Wortlisten = Zungenbrecher.Select(w => w.Split(' '));  
  
// Einfacher geht es mit dem SelectMany-Operator,  
// der eine flache Liste erzeugt und nur eine Schleife  
// erforderlich macht. Durch den zusätzlichen Einsatz des  
// Select-Operators kann neben dem laufenden Index des Zungenbrechers auch  
// auf den laufenden Index der Wörter zugegriffen werden.  
var flacheWortliste =  
    Wortlisten.SelectMany((z,n) =>  
        z.Select((w,m) => string.Format("Brecher# {0}, Wort# {1}: {2}",n,m,w)));  
// Ausgabe  
foreach (var Wort in flacheWortliste)  
    Debug.WriteLine(Wort);
```

In der Ausgabe ist zu erkennen, in welchem Zungenbrecher das Wort an wievielter Stelle steht. Die ersten sieben Zeilen der Ausgabe sehen so aus:

```
Brecher# 0, Wort# 0: Fischers
Brecher# 0, Wort# 1: Fritze
Brecher# 0, Wort# 2: fischt
Brecher# 0, Wort# 3: frische
Brecher# 0, Wort# 4: Fische.
Brecher# 1, Wort# 0: Brautkleid
Brecher# 1, Wort# 1: bleibt
```

Überladung 3:

Als Beispiel für die dritte Überladung dient ein kleines Array mit Objekten, die sowohl den Namen als auch eine Liste mit Automarken von berühmten Autobesitzern speichert. Mittels `SelectMany` wird daraus eine flache Liste. Der zweite Lambda-Ausdruck wird dabei verwendet, um Name und Marke miteinander in Beziehung zu setzen.

```
// Ein kleine Liste berühmter Autofahrer
AutoFahrer[] AutoFahrer = {
    new AutoFahrer { Name="Percy Pickwick",
        Autos = new List<string>{ "MG TD" } },
    new AutoFahrer { Name="James Bond",
        Autos = new List<string>{ "BMW Z8", "Lotus Esprit", "AMC Hornet" } }
};

// Mit dem SelectMany-Operator wird daraus eine flache Liste, die
// für jede Fahrer/Auto-Kombination ein anonymes Objekt enthält
var Fahrerliste =
    AutoFahrer.SelectMany(Fahrer => Fahrer.Autos, (Fahrer, Auto) =>
        new { Fahrer.Name, Auto });

// Alternativ kann natürlich auch z.B. die String.Format()-Methode verwendet
// werden, um eine entsprechende Liste von Zeichenketten zu erstellen
IEnumerable<string> Fahrerliste2 =
    AutoFahrer.SelectMany(Fahrer => Fahrer.Autos, (Fahrer, Auto) =>
        string.Format("{0} fährt {1}", Fahrer.Name, Auto ));

// Ausgabe
foreach (var s in Fahrerliste)
    Debug.WriteLine (s);

// Klasse zum Speichern der Fahrer und einer Liste von Autos
```

```
class AutoFahrer
{
    public string Name { get; set; }
    public List<string> Autos { get; set; }
}
```

Die Ausgabe besteht aus vier Zeilen mit dem für anonyme Objekte typischen Aufbau:

```
{ Name = Percy Pickwick, Auto = MG TD }
{ Name = James Bond, Auto = BMW Z8 }
{ Name = James Bond, Auto = Lotus Esprit }
{ Name = James Bond, Auto = AMC Hornet }
```

3.2 Filteroperatoren

Sollen nicht alle Elemente einer Sequenz verwendet werden, sondern nur solche, die einem bestimmten Filter entsprechen, so benutzt man Filteroperatoren. Der Kern eines Filters ist das Prädikat, das vom Typ `bool` ist und daher entweder wahr oder falsch sein kann. Für LINQ in Verbindung mit C# bedeutet dies, dass gewöhnliche C#-Bedingungen zum Einsatz kommen, die entweder `true` oder `false` liefern. Aus diesem Grund finden auch die üblichen Regeln über Auswertungsprioritäten Anwendung, wie sie für `if`-Abfragen oder `while/do..while`-Schleifen der Fall sind.

3.2.1 Where

Der einzige Standardfilteroperator steht sowohl in der LINQ-Syntax (für die erste Überladung) als auch als Methode zur Verfügung.

Überladungen

Für diesen Operator existieren zwei Überladungen.

Überladung 1:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);
```

Überladung 2:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate);
```

Ausnahmen

Es wird eine Ausnahme vom Typ `ArgumentNullException` ausgelöst, wenn **source** oder **predicate** lediglich null sind.

Beispiele

Überladung 1:

Als Beispiel für die erste Überladung dient eine Abfrage auf alle laufenden Prozesse, deren Prozessname mit einem , W' beginnt und die mehr als fünf Threads verwenden.

```
// Namensraum für Zugriff auf Prozesse einführen
using System.Diagnostics;

// Alle laufenden Prozesse auslesen
Process[] procList = Process.GetProcesses();

// Es sollen alle Prozesse ausgewählt werden,
// die mit einem 'W' beginnen und die mehr als fünf
// Threads verwenden.
IEnumerable<Process> ProcList1 = from proc in procList
                                where proc.ProcessName.StartsWith("W") &&
                                       proc.Threads.Count > 5
                                select proc;

// Alternativ kann auch die Methoden-Syntax verwendet werden.
// In diesem Fall kann der Select-Operator entfallen, da hier sowieso
// nur die Process-Objekte unverändert ausgewählt werden.
IEnumerable<Process> ProcList2 = (procList.Where(proc =>
    proc.ProcessName.StartsWith("W") && proc.Threads.Count > 5));

// Ausgabe
foreach (Process proc in ProcList1)
    // Statt der Ausgabe könnte auch alles andere unternommen werden,
    // was mit einem Process-Objekt möglich ist, z.B. beenden via Kill().
    Debug.Print("{0}: {1} (Threads: {2})",
        proc.Id, proc.ProcessName, proc.Threads.Count);
```

Die Ausgabe hängt natürlich davon ab, was derzeit auf dem Computer ausgeführt wird. Sie könnte in etwa so aussehen:

```
3496: WmiPrvSE (Threads: 6)
5724: WINWORD (Threads: 9)
```

Überladung 2:

In der zweiten Überladung steht zusätzlich ein laufender, nullbasierter Index zur Verfügung. Dieser wird im Beispiel dafür genutzt, eine Liste der laufenden Prozesse auf maximal fünf zu begrenzen.

Hinweis

Der gleiche Effekt kann mittels des Take-Operators erreicht werden. Für Details siehe Abschnitt 3.10.1 *Take*.

```
// Namensraum für Zugriff auf Prozesse einführen
using System.Diagnostics;

// Alle laufenden Prozesse auslesen
Process[] procList = Process.GetProcesses();

// Hier steht nur die Methoden-Syntax zur Verfügung.
// In diesem Fall kann der Select-Operator entfallen, da hier sowieso
// nur die Process-Objekte unverändert ausgewählt werden.
IEnumerable<Process> = (procList.Where((proc, index) => index < 5));

// Mit einem Select-Operator könnte die Abfrage alternativ so aussehen.
// Die Sequenz enthält in diesem Fall dann nur noch String-Objekte.
IEnumerable<string> ProcList2 = (procList.Where((proc, index) => index < 5)).
    Select(proc => string.Format(
        "{0}: {1} (Threads: {2})", proc.Id, proc.ProcessName, proc.Threads.Count));

// Ausgabe
foreach (Process proc in ProcList1)
    // Auch hier kann statt der Ausgabe alles andere unternommen werden,
    // was mit einem Process-Objekt möglich ist, z.B. beenden via Kill().
    Debug.Print(
        "{0}: {1} (Threads: {2})", proc.Id, proc.ProcessName, proc.Threads.Count);
```

Die Ausgabe entspricht der, die wir oben für die erste Überladung gezeigt haben und beschränkt sich im Übrigen auf die ersten fünf Prozesse, welche die `Process.GetProcesses()`-Methode liefert.

```
3740: LINQ.vshost (Threads: 13)
3148: svchost (Threads: 14)
4004: winampa (Threads: 1)
4720: iexplore (Threads: 16)
2748: SearchIndexer (Threads: 22)
```

3.2.2 OfType

Dieser Operator filtert die Elemente einer Sequenz nach einem bestimmten Datentyp. Alle Elemente, die von diesem Typ sind oder von diesem abgeleitet wurden, werden übernommen. Alternativ kann statt eines Typs auch eine Schnittstelle angegeben werden, so dass alle Elemente, die diese implementieren, übernommen werden.

Signatur

Dieser Operator ist nicht überladen. Es existiert nur eine Signatur:

```
public static IEnumerable<TResult> OfType<TResult>(
    this IEnumerable source);
```

Ausnahmen

Es wird eine Ausnahme vom Typ `ArgumentNullException` ausgelöst, wenn **source** lediglich `null` ist.

Beispiel

Das Beispiel zeigt, wie aus einem gemischten Array vom Typ `object` alle Elemente gefiltert werden, die vom Typ `string` sind.

```
// Gemischtes Array erzeugen
object[] DiesUndDas = { 1, "Ronald" ,true, "Ruck"};

// Sequenz mit allen Zeichenketten erstellen
IEnumerable<string> zeichenketten = DiesUndDas.OfType<string>();

// Ausgabe
foreach (string s in zeichenketten)
    Debug.WriteLine(s);
```

Die Ausgabe besteht aus den zwei Zeichenketten und hat eine auffällige Ähnlichkeit mit einer gewissen Ente:

```
Ronald
Ruck
```

3.3 Sortierungsoperatoren

Sollen Sequenzen in einer bestimmten Reihenfolge vorliegen, so müssen sie zuvor sortiert werden. Zu diesem Zweck stehen Sortierungsoperatoren zur Verfügung. Mit ihnen können beliebige Sortierungen durchgeführt oder die Reihenfolge der Elemente in einer Sequenz umgekehrt werden.