LINQ – Teil 2

Ingo Köster

Diplom-Informatiker (FH)

LINQ Abfrageoperatoren Teil 1

Operatortyp	Operator
Aggregatoperatoren	`Aggregate, Average, Count, LongCount, Min, Max, Sum`
Casting-Operatoren	`Cast, OfType, ToArray, ToDictionary, ToList, ToLookup, ToSequence`
Elementoperatoren	`DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault`
Gleichheitsoperatoren	`EqualAll`
Sequenzoperatoren	`Empty, Range, Repeat`
Gruppierungsoperatoren	`GroupBy`
Join-Operatoren	`Join, GroupJoin`

LINQ Abfrageoperatoren Teil 2

Operatortyp	Operator
Sortieroperatoren	`OrderBy, ThenBy, OrderByDescending, ThenByDescending, Reverse`
Aufteilungsoperatoren	`Skip, SkipWhile, Take, TakeWhile`
Quantifizierungsoperatoren	`All, Any, Contains`
Restriktionsoperatoren	`Where`
Projektionsoperatoren	`Select, SelectMany`
Set-Operatoren	`Concat, Distinct, Except, Intersect, Union`

Select

- Projiziert jedes Element der Sequenz mittels eines Lambda-Ausdrucks in ein neues Format

```
string[] names = { "Tom", "Hans", "Peter" };

var grosseNamen = names.Select(n => n.ToUpper());

var grosseNamen = from n in names
                  select n.ToUpper();
```

- Ergebnismenge
  - Ist vom Typ `IEnumerable`
  - Kann mittels `foreach` durchlaufen werden
  - Kann in Array oder Collection umgewandelt werden

Take & Skip

- Take
  - Gibt angegebene Anzahl von Elementen ab dem Anfang einer Sequenz zurück
- Skip
  - Überspringt angegebene Anzahl von Elementen einer Sequenz und gibt die übrigen Elemente zurück

- Beispiele
  - `int[] zahlen = { 14, 42, 23, 19, 18, 47 };`
  - `var dieErstenDrei = zahlen.Take(3);  // 14,42,23`
  - `var dieletztenZwei = zahlen.Skip(4);  // 18,47`

Elementoperatoren

```
int[] zahlen = { 14, 42, 23, 19, 18, 47 };

zahlen.First();          // 14
zahlen.Last();           // 47
zahlen.ElementAt(1);     // 42

// Erste ungerade Zahl
zahlen.First(x => x % 2 == 1);     // 23
```

- Werfen Exceptions, wenn keine Elemente vorhanden sind
- `FirstOrDefault`, `LastOrDefault`, etc. liefern default-Wert anstatt einer Exception

Elementoperatoren

```
› int[] zahlen = { 14, 42, 23, 19, 18, 47 };

› zahlen.Single(x => x > 45);
```

- Gibt das einzige Element einer Sequenz zurück, dass die angegebene Bedingung erfüllt
- Löst eine Ausnahme aus, wenn mehrere Elemente die Bedingung erfüllen
  - Alternative: `SingleOrDefault`

Where

› Filtert eine Sequenz von Werten nach einem Prädikat

```
› int[] zahlen = { 14, 42, 23, 19, 18, 47 };

› var kleinerZwanzig = zahlen.Where(x => x < 20);

› string[] namen = { "Tom", "Hans", "Peter", "Tim", "Arnold" };

› var tUndM = namen.Where(n => Regex.IsMatch(n, @"t.m",
  RegexOptions.IgnoreCase)); // Tom,Tim
```

Aggregatoperatoren

```
› int[] zahlen = { 14, 42, 23, 19, 18, 47 };

› double durchschnitt = zahlen.Average(); // 27,1666

› int anzahl = zahlen.Count();            // 6

› int kleinstes = zahlen.Min();           // 14

› int größtes = zahlen.Max();             // 47

› int summe = zahlen.Sum();               // 163
```

OrderBy

```
› int[] zahlen = { 14, 42, 23, 19, 18, 47 };

› var sortiert = zahlen.OrderBy(x => x);



› string[] namen = { "Tom", "Hans", "Peter", "Tim", "Arnold"
  };

› var nachLänge = namen.OrderBy(n => n.Length);
```

Sortieren

- Absteigend
  - `OrderByDescending anstatt OrderBy`

- Zwei Sortierkriterien (erst Länge, dann Alphabet):
  -
```
string[] names = { "Tom", "Hans", "Peter", "Kim", "Albrecht",
"Arnold", "Bodo", "Udo" };
```
  -
```
var sorted = names.OrderBy(n => n.Length).ThenBy(n => n);
```

- Ergebnis: Kim, Tom, Udo, Bodo, Hans, Peter, Arnold, Albrecht

Sortieren - Abfragesyntax

```
int[] zahlen = { 14, 42, 23, 19, 18, 47 };
var sortiert = from x in zahlen
               orderby x ascending
               select x;

// ascending ist Standard
// descending für absteigende Sortierung
string[] names = { "Tom", "Hans", "Peter", "Kim", "Albrecht", "Arnold",
"Bodo", "Udo" };

var sorted = from n in names
             orderby n.Length, n
             select n;
```