LINQ

Language Integrated Query

LINQ

LINQ = „**L**anguage **In**tegrated **Q**uery“

LINQ operiert auf Objekten des Typs **`IEnumerable<>`**.

D.h. LINQ ist anwendbar auf **Collections** aller Art:

- Arrays
- List
- Dictionary
- string
- …
- als LINQ to Entities auch auf Datenbank-Tabellen (genauer: auf Klassen mit Interface **`IQueryable<>`**)

LINQ

LINQ besteht aus einer Reihe von verschiedenen Methoden, mit denen man Collections bearbeiten kann

- **Filtern**, z.B. Where, Distinct, Skip, …
- **Projizieren**: Select
- **Sortieren**: OrderBy, Reverse, …
- **Konvertieren**: ToList, ToArray, …
- **Auswählen**: First, Last, …
- **Aggregieren**: Max, Min, Sum, …

Spracherweiterungen

LINQ verwendet folgende besprochenen C#-Spracherweiterungen:

- **Extensionmethods**: Where, Select, usw. sind als Erweiterungsmethoden implementiert
- **Generics**: Alle diese Methoden sind natürlich generisch implementiert
- **Lambda-Ausdrücke**: Auswahl, Sortierung,… in den Listen erfolgt praktisch immer durch Lambdas
- **Anonyme Typen**: damit kann man bei der Auswahl (Select) beliebige Strukturen erstellen
- **var**: daher wird als Returntyp häufig var verwendet

Schreibweisen

Es gibt zwei Möglichkeiten, LINQ-Queries zu codieren

- **Lambda-Syntax**
- Comprehension-Syntax oder kurz **Query-Syntax**

Hinweise:

- Ausdrücke in Query-Syntax werden vom Compiler immer in Lambda-Syntax umgewandelt.
- Man kann Lambda-Syntax und Query-Syntax auch mischen.

Lambda-Syntax

- Wie der Name schon sagt, werden Erweiterungsmethoden mit Lambda-Ausdrücken verwendet.
- Beispiel:

```
string[] names = { "Franz", "Udo", "Hans", "Susi", "Tom" };
var aNames = names
   .Where(s => s.Contains('a'))
   .OrderBy(s => s)
   .Select(s => s.ToUpper());
foreach (string s in aNames)
{
   Console.WriteLine(s);
}
```

```
names
   .Where(s => s.Contains('a'))
   .OrderBy(s => s)
   .Select(s => s.ToUpper())
   .ToList()
   .ForEach(Console.WriteLine);
```

- Man beginnt mit der Collection, filtert, sortiert und projiziert (=selektiert).
- Die Funktion, wie gefiltert, sortiert und projiziert wird, wird als Lambda-Ausdruck angegeben.
- Die Daten „fließen“ durch die Operatorenkette von links nach rechts.
- Die Eingabesequenz wird dabei nie verändert!

Query-Syntax

- Ist eine syntaktische Abkürzung für Lambda-Syntax.
- Sieht aus wie eine **auf den Kopf gestellte SQL-Abfrage**.
- Obiges Beispiel in Query-Syntax:

```
var aNames = from s in names
             where s.Contains('a')
             orderby s
             select s.ToUpper();
foreach ( string s in aNames )
{
    Console.WriteLine(s);
}
```

- beginnt mit **from**
- mit from wird eine **Laufvariable** festgelegt (ähnlich einer Laufvariable bei foreach)
- endet mit **select** (oder group)

Gemischte Syntax

- In bestimmten Situation kann eine Mischung aus Lambda- u. Query-Syntax die beste Wahl sein.
- Teilweise notwendig, weil **manche Operatoren nur in Lambda-Syntax** verfügbar sind.
- Beispiel:

```
int count = (from n in names
             where n.Contains("a")
             select n
             ).Count();
```

Lambda oder Query?

Vorteil Query-Syntax

- Sieht aus wie SQL
- keine Klammern

Vorteil Lambda Syntax:

- verwendet wie gewohnt Funktionen
- man kann sich (teilweise) abschließendes Select sparen
- ist gebräuchlicher

```
names
   .Where(x => x.Contains('a'))
   .OrderBy(x => x)
   .Select(x => x.ToUpper())
   .ToList()
   .ForEach(Console.WriteLine);
```

Best practice

- **Lambda** Syntax verwenden
- pro Schlüsselwort eine **neue Zeile** beginnen
- Variable im Lambda-Ausdruck immer **x**

Verzögerte Ausführung

Eine Abfrage wird nicht bei der Erstellung ausgeführt.

Tatsächlich wird eine LINQ-Abfrage ausgeführt, bei

- Operatoren, die ein **einzelnes Element** zurückgeben, z.B. First
- Operatoren, die einen **Skalarwert** zurückgeben, z.B. Count
- **Konvertierungsoperatoren**, z.B. ToArray, ToList
- Enumeration über die Abfrage, also zum Zeitpunkt der **foreach**-Schleife

Verzögerte Ausführung

Beispiel

```
var numbers = new List<int>() { 1, 2, 3, 4 };
var timesTen = numbers.Select(n => n*10);
numbers.Add(5);
foreach (var x in timesTen)
{
    Console.WriteLine(x);
}
```

Ausgabe: **10, 20, 30, 40, 50**

Verzögerte Ausführung

Aber:

```
var numbers = new List<int>() { 1, 2, 3, 4 };
var timesTen = numbers.Select(n => n*10).ToList();
numbers.Add(5);
foreach (var x in timesTen)
{
    Console.WriteLine(x);
}
```

Ausgabe: **10, 20, 30, 40**

Verkettung

- Mit jedem Operator entsteht eine unterschiedliche Sequenz.
- Es entsteht also eine Kette von Dekoratoren.

```
IEnumerable<int> source = new int[] { 5, 12, 3 };
var filtered = source   .Where(n => n < 10);
var sorted   = filtered.OrderBy(n => n);
var result   = sorted   .Select(n => n * 2);
foreach (var x in result)
{
    Console.WriteLine(x);
}
```

- Die Anzahl der Elemente kann dabei nie größer werden!
- Auch hier gilt: Ausgeführt wird die Abfrage erst bei Iteration bzw. **`ToList()`**.

Anonyme Typen

- Bei **`Select`** kann man sehr einfach die selektierten Werte als anonyme Typen zu neuen Klassen kombinieren.

```
var erNames = fullNames
              .Where(x => x.EndsWith("er"))
               .OrderBy(x => x.Length)
               .Select(x => new
               {
                 firstName = x.Split(' ')[0],
                 lastName = x.Split(' ')[1].ToUpper()
               });
foreach (var name in erNames)
{
  Console.WriteLine(name);
}
```

DRY

```
var erNames = fullNames
              .Where(x => x.EndsWith("er"))
               .OrderBy(x => x.Length)
               .Select(x => new
               {
                 firstName = x.Split(' ')[0],
                 lastName = x.Split(' ')[1].ToUpper()
               });
foreach (var name in erNames)
{
  Console.WriteLine(name);
}
```

- Im vorigen Beispiel wurde Code kopiert (x.Split()).
- Das kann man durch ein **zusätzliches `Select`** vermeiden

```
var erNames = fullNames
              .Where(x => x.EndsWith("er"))
               .OrderBy(x => x.Length)
               .Select(x => x.Split(' '))
               .Select(x => new
               {
                 firstName = x[0],
                 lastName = x[1].ToUpper()
               });
```

ForEach

- Die Methode **`ForEach`** ist nicht für IEnumerable, sondern nur für **`List<T>`** definiert.
- Damit kann man Code oft verkürzen

```
fullNames
   .Where(x => x.EndsWith("er"))
   .OrderBy(x => x.Length)
   .Select(x => x.Split(' '))
   .Select(x => new
   {
     firstName = x[0],
     lastName = x[1].ToUpper()
   })
   .ToList()
   .ForEach(x => Console.WriteLine(x));
```

```
});
.ToList()
.ForEach(Console.WriteLine);
```