LINQ

Vincent Uhlmann
IT-Akademie Dr. Heuer GmbH

LINQ

- LINQ steht für Language Integrated Query
- Eine in C# integrierte Query Language
- Verwendet eine Syntax ähnlich zu SQL
- Dient der Abfrage und Manipulation von Daten
- Bietet eine einheitliche Syntax für unterschiedliche Datenquellen (Arrays, Liste, SQL-Datenbanken..)
- Kann als Query-Syntax oder Methoden-Syntax verwendet werden
- Die Daten werden erst bei der Ausführung der Abfrage abgerufen

QUERY-SYNTAX

- Ähnelt SQL
- Bietet klare Syntax für komplexe Abfragen
- Unterstützt die Verwendung von Lambda-Ausdrücken

- Keywords wie:
- from
- where
- orderby
- select

QUERY-SYNTAX

- 1. Datenquelle abrufen
- 2. Abfrage erstellen
- 3. Abfrage ausführen

```
// Datenquelle
int[] ints = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Abfrage erstellen
var equalInts = from num in ints where num % 2 == 0 select num;

// Abfrage ausführen
foreach (var item in equalInts)
    Console.WriteLine(item);
```

METHODEN- SYNTAX

- Nutzt Methodenaufrufe
- Für einfache Abfragen, wie where, select und orderBy, kann die Methodensyntax einfacher und direkter sein als bei der Query-Syntax
- Ermöglicht das Verketten von Methodenaufrufen
- Auch hier werden Lambda-Ausdrücke verwendet

- Where(lambda)
- Select(lambda)
- OrderBy(lambda)

METHODEN- SYNTAX

- 1. Datenquelle abrufen
- 2. Abfrage erstellen
- 3. Abfrage ausführen

```
// Datenquelle
int[] ints = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

// Abfrage erstellen
var equalInts = ints.Where(x  => x % 2 == 0);

// Abfrage ausführen
foreach (var item in equalInts)
    Console.WriteLine(item);
```

LINQ FEATURES

- Einige der in C# genutzten Features werden durch LINQ hinzugefügt

- Query Expressions
- Extension Methods
- Die Typableitung durch var
- Objekt- und Sammlungsinitialisierer
- Anonyme Typen

OBJEKT- UND SAMMLUNGSINITIALISIERER

- Eigenschaften eines Objektes direkt beim Erstellen des Objektes initialisieren

```
public class Tier
{
    public string Art { get; set; }
    public int Age { get; set; }
}
```

- Sammlungen direkt beim Erstellen initialisieren

```
Tier tier = new Tier { Art = "Hund", Age = 10 };

List<Tier> list = new List<Tier>
{
    new Tier { Art = "Hund", Age = 10 },
    new Tier { Art = "Katze", Age = 15 }
};
```

ANONYME TYPEN

- Ermöglichen das Erstellen schreibgeschützter Typen in einem Objekt, ohne expliziten Typen
- Die Typen der Eigenschaften werden vom Compiler abgeleitet
- Der Typname wird vom Compiler generiert und ist im Code nicht verfügbar

```
var v = new {
    Amount = 108,
    Message = "Hello"
};

Console.WriteLine(v.Amount + v.Message);
```

LINQ OPERATOREN

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 OPERATOREN

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

BEISPIEL KLASSE

- Gegeben ist folgende Klasse Person

```
public class Person
{
    public string Name { get; set; } = string.Empty;
    public int Alter { get; set; }

    public Person(string name, int alter)
    {
        Name = name;
        Alter = alter;
    }
}
```

BEISPIEL WHERE

- Filtert Elemente einer Sammlung basierend auf einer Bedingung

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

IEnumerable<Person> newList = persons.Where(x => x.Alter % 2 == 0);
```

BEISPIEL SELECT

- Transformiert Elemente einer Sammlung in eine neue Form

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

IEnumerable<string> namen = persons.Select(x => x.Name);
```

BEISPIEL SUM

- Berechnet die Summe einer Sammlung

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

var summeAlter = persons.Sum(x => x.Alter);
```

BEISPIEL COUNT

- Zählt die Anzahl der Elemente in einer Sammlung, die eine bestimmte Bedingung erfüllen

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

var personenÜ30Count = persons.Count(x => x.Alter > 30);
```

BEISPIEL ORDERBY

- Sortiert die Elemente einer Sammlung basierend auf einem oder mehreren Schlüsseln

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

var sortierteListe = persons.OrderBy(x => x.Alter);
```

BEISPIEL FIRST

- Gibt das erste Element einer Sammlung zurück, auf das eine bestimmte Bedingung zutrifft
- Löst eine InvalidOperationException aus wenn kein passendes Element vorhanden ist

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

var item = persons.First(x => x.Alter == 24);
```

BEISPIEL FIRSTORDEFAULT

- Gibt das erste Element einer Sammlung zurück, auf das eine bestimmte Bedingung zutrifft
- Gibt den Standardwert des Elementtyps zurück, anstatt eine InvalidOperationException auszulösen

```
List<Person> persons = new List<Person>();
persons.Add(new Person("Peter", 50));
persons.Add(new Person("Franz", 24));
persons.Add(new Person("Marie", 25));

var item = persons.FirstOrDefault(x => x.Alter == 24);
```