Teil L

Spracherweiterungen und LINQ

In diesem Teil:

Kapitel 45

Spracherweiterungen von C# 3.5

In diesem Kapitel:

Die Spracherweiterungen von C# 3.5 weisen in eine funktionale Richtung der Sprache C#, die vorwiegend in LINQ verwendet wird. Die *lokale Typinferenz* etwa erlaubt es, eine Variable als `var` zu definieren und anhand des Initialwertes typisieren zu lassen. Die *Lambda-Ausdrücke* stellen einen Zeiger-Parameter zur Verfügung, der eine leichtere Handhabung von anonymen Methoden erlaubt. Und die Erweiterungsmethoden ermöglichen es, die Kompatibilität zwischen Klassen in unterschiedlichen Bibliotheken zu gewährleisten, wenn diese um zusätzliche Methoden erweitert werden müssen. Man sieht sehr schnell … es haben sich in der aktuellen Version interessante Erweiterungen der Sprache C# ergeben.

Lokale Typinferenz

Die *lokale Typinferenz* erinnert ein wenig an die Skriptsprache VB Script, in der es möglich war, eine nicht typisierte Variable zu erzeugen. In der Tat kann die Typinferenz nur als lokale Variable deklariert werden und vereinfacht dabei u. a. die Kodierung. Die Entscheidung über die korrekte Typisierung wird dem Compiler überlassen. Im Gegensatz zu expliziten Typen wie beispielsweise `int` oder `string` wird bei der lokalen Typinferenz eine Variable mit dem Schlüsselwort `var` definiert und deklariert. Der folgende Code zeigt eine Deklaration einer Variablen `i` über die lokale Typinferenz.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Spracherweiterung
{
    /// <summary>
    /// Spracherweiterungen von C# 3.5
    /// </summary>
    public class Program
    {
        public static void Main(string[] args)
        {
            var i = 7; //die literale Konstante 7 deklariert die Variable i als Integer (int)

            Console.WriteLine(i.GetType().ToString());

            Console.ReadLine();
        }
    }
}
```

Die literale Konstante `7` ist vom Typ `int` bzw. `Int32` und somit ist die Variable `i` ebenfalls vom Typ `int`. Die Ausgabe dieses Codes ist der aktuelle Typ der Variable `i`: *System.Int32*. Im Gegensatz zu älteren Programmiersprachen wie zum Beispiel Visual Basic 6.0 ist eine Deklaration über die lokale Typinferenz eine typensichere Deklaration.

Lokale Typinferenzen können nicht explizit mit `null` deklariert werden und müssen initialisiert werden, da sonst der Compiler keine Typisierung vornehmen kann. Die beiden folgenden Anweisungen führen deshalb zu einem Compiler-Fehler.

```
var withNull = null;
var withoutInit;
```

Wofür man lokale Typinferenz benötigt, wird im Abschnitt »Anonymer Typ« und in Kapitel 46 ersichtlich.

HINWEIS Eine Variable, die über die lokale Typinferenz deklariert wird, kann nur als lokale Variable bzw. innerhalb einer Methode deklariert werden. Versuche, die lokale Typinferenz für Klassenvariablen oder Parameter zu verwenden, resultieren in Compiler-Fehlermeldungen.

Erweiterungsmethoden

Mithilfe von Erweiterungsmethoden ist es möglich, Klassen und Strukturen nachträglich zu erweitern, selbst wenn es sich um `sealed`-Klassen oder Strukturen wie `System.Int32` handelt. Erweiterungsmethoden sind statische Methoden, die innerhalb einer statischen Klasse implementiert werden. Erweiterungsmethoden die eine instanziierte Klasse erweitern, werden als Instanz-Methoden aufgerufen, obwohl sie statisch deklariert wurden. Der folgende Code erweitert die `Int32`-Struktur des C#-Datentyps `int`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Erweiterungsmethoden
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int i = -5;

            Console.WriteLine(i.AbsoluteValue().ToString());
            Console.ReadLine();
        }
    }

    public static class _Int32_
    {
        public static int AbsoluteValue(this int me)
        {
            return System.Math.Abs(me);
        }
    }
}
```

Hier wird dem Datentyp `int` durch die Klasse `_Int32_` eine weitere Methode hinzugefügt, die den absoluten Wert einer Zahl zurückgeben soll. Wenn eine Methode als Erweiterungsmethode definiert wird, muss vor dem ersten Parameter das Schlüsselwort `this` stehen, gefolgt von der Struktur oder Klasse, die erweitert werden soll. In unserem Beispiel wird der Datentyp `int` bzw. die `Int32`-Struktur erweitert, weswegen `this int me`

in der Parameterliste der Methode `AbsoluteValue()` steht. Der erste Parameter wird beim Aufruf nicht mitgeführt.

Möchte man einer Erweiterungsmethode Argumente übergeben, gibt man weitere Parameter in der Parameterliste an.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Erweiterungsmethoden
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int i = -5;

            Console.WriteLine(i.AbsoluteValue().ToString());
            Console.WriteLine(i.AbsoluteValuePlus(10).ToString());
            Console.ReadLine();
        }
    }

    public static class _Int32_
    {
        public static int AbsoluteValue(this int me)
        {
            return System.Math.Abs(me);
        }

        public static int AbsoluteValuePlus(this int me, int plus)
        {
            int i = System.Math.Abs(me) + plus;
            return i;
        }
    }
}
```

Erweiterungmethoden können nicht nur für Strukturen, sondern auch für jede Art von Klassen geschrieben werden.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace _02Erweiterungsmethoden
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```
            ExtendedCalc calculator = new ExtendedCalc();
            Console.WriteLine(calculator.Add(3, 4).ToString());
            Console.WriteLine(calculator.Sub(3, 4).ToString());
            Console.ReadLine();
        }
    }

    public class SimpleCalc
    {
        public int Add(int i, int j)
        {
            return i + j;
        }
    }

    public class ExtendedCalc : SimpleCalc
    {
    }

    //Erweiterung der Basisklasse SimpleCalc
    public static class _SimpleCalc_
    {
        public static int Sub(this SimpleCalc simple, int i, int j)
        {
            return i - j;
        }
    }
}
```

Listing 45.1 Erweiterungsmethoden einer Klasse (aus dem Projekt Erweiterungsmethoden)

Die statische Klasse `_SimpleCalc_` definiert hier die statische Erweiterungsmethode `Sub()`, die aber im Hauptprogramm anders als vielleicht erwartet, als Instanzmethode aufgerufen wird.

Objektinitialisierer

Über einen Objektinitialisierer ist es möglich, Felder und Eigenschaften einer Klasse ohne einen Konstruktor zu initialisieren. Diese Art von Objektinitialisierung ist z.B. sinnvoll, wenn keine Konstruktorlogik benötigt wird. Die Reihenfolge der Initialisierung ist beliebig, da die Initialisierung immer über ein bezeichnetes (Feldname oder Eigenschaftsname) Tupel stattfindet.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace _Objektinitialisierer
{
    public class Program
    {
        public static void Main(string[] args)
```

```
        {
            Person p = new Person { Firstname = "Aaron", Lastname = "Strasser", Age = 3, Weight = 70 };

            Console.WriteLine(p.Age.ToString());
            Console.WriteLine(p.Firstname);
            Console.WriteLine(p.Lastname);
            Console.WriteLine(p.Weight);

            Console.ReadLine();
        }
    }

    public class Person
    {
        public int Age;
        public string Firstname;
        public string Lastname;
        private int weight;

        public int Weight
        {
            get { return weight; }
            set { weight = value; }
        }
    }
}
```

Listing 45.2 Objektinitialisierer (aus dem Projekt Objektinitialisierung)

Anonymer Typ

In der Programmierung ist es des Öfteren sinnvoll, einfache Klassen (Hilfsklassen oder Eigenschaftsklassen) zu erzeugen. Hier bietet sich der Einsatz von anonymen Typen bzw. Klassen an, deren Konstrukt namenslos verwendet wird. Der Compiler vergibt bei der Deklaration einen zufälligen Namen, um den Typ bzw. die Instanz erzeugen zu können. Somit ist der Typ von außen nicht direkt zugreifbar. Um eine Klasse `Person` wie aus dem vorangehenden Abschnitt durch einen anonymen Typen zu simulieren, müssen Sie lediglich eine `var`-Variable über einen Objektinitialisierer erzeugen.

```
var p = new { Firstname="Aaron", Lastname="Strasser", Age=3, Weight=70 };
```

Das Objekt `p` kann danach wie in Listing 45.3 verwendet werden.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace AnonymeTypen
{
    public class Program
    {
```

```
        public static void Main(string[] args)
        {
            var p = new { firstname = "Aaron", lastname = "Strasser", age = 3, Weight = 70 };

            Console.WriteLine(p.age.ToString());
            Console.WriteLine(p.firstname.ToString());
            Console.WriteLine(p.lastname.ToString());

            Console.ReadLine();
        }
    }
}
```

Listing 45.3 Anonyme Typen (aus dem Projekt AnonymeTypen)

Beachten Sie aber, dass im Falle des anonymen Typs hinter `Weight` keine Eigenschaft, sondern nur ein einfaches Feld steht.

Der Typ wird bei anonymen Typen über den Objektinitialisierer definiert, sodass Objekte, die mit gleichem Objektinitialisierer erzeugt wurden, demselben Typ angehören und einander zuweisbar sind.

```
var p1 = new { firstname="Aaron", lastname="Frank", age=4, Weight=70 };
var p2 = new { firstname="Angela", lastname="Frank", age=34, Weight=60 };
...
p1 = p2;
...
Console.WriteLine(p1.firstname.ToString()); //Angela
```

ACHTUNG Damit zwei Objekte, die mit Objektinitialisierern erzeugt wurden, demselben Typ angehören, müssen die Initialisierungsvariablen in beiden Fällen in derselben Reihenfolge aufgelistet werden.

HINWEIS In Objektinitialisierern für anonyme Typen ist es nicht möglich, einem Feld `null` zuzuweisen, da der Compiler dann nicht mehr in der Lage ist, den Typ des Feldes zu bestimmen.

Lambda-Ausdrücke

Mit den Delegaten aus C# 1.0 und 1.1 wurde der Sprache eine Möglichkeit geben, Zeiger auf Code erzeugen zu können. In C# 2.0 wurde durch die Einführung der anonymen Methoden eine Übergabe von Code-Zeigern als Argumente an Methoden eingeführt. Mit der Einführung von C# 3.5 bekommt die unhandliche Syntax der anonymen Methoden eine klarere und einfachere Syntax. Des Weiteren werden durch die Lambda-Ausdrücke mehr Möglichkeiten geschaffen, die in diesem Kapitel erläutert werden.

Rückblick

Leser, die mit dem Konzept der Delegaten noch nicht so vertraut sind, sollten vorab das Kapitel 18 »Delegaten und Ereignisse« durchlesen, um den harten Weg vom Delegaten zum Lambda-Ausdruck besser verstehen zu können. Fortgeschrittene, die bereits die Konzepte der Delegaten in C# 1.0 bis C# 2.0 eingesetzt haben, können hier sofort weiterlesen und ihr Wissen auffrischen.

Delegaten erlauben dem Entwickler, einen Zeiger auf Code zu erzeugen. Dieser Code kann jede Methode sein, die der Delegaten-Definition entspricht.[1]

```
namespace Lambda01
{
    delegate int myDelegate(int i, int j);

    public class Program
    {
        public static void Main(string[] args)
        {
            SmartCalc calc = new SmartCalc();

            myDelegate add = new myDelegate(calc.Add);
            Console.WriteLine("{0}", add(3, 4).ToString());

            Console.ReadLine();
        }
    }

    public class SmartCalc
    {
        public int Add(int i, int j)
        {
            return i + j;
        }
    }

}
```

Listing 45.4 Zeiger auf Code über Delegaten (aus dem Projekt Lambda01)

Listing 45.4 zeigt die Delegaten-Definition `delegate int myDelegate(int i, int j)` mit zwei Parametern vom Typ `int`. Die Klasse `SmartCalc` enthält eine Methode `Add()`, die der Delegaten-Definition entspricht. Die Anwendung erzeugt in `Main()` eine Instanz der Klasse `SmartCalc` und übergibt der Instanz des Delegaten den Zeiger auf die Methode `Add()` der Klasseninstanz `calc`.

Mit der Einführung anonymer Methoden in C# 2.0 war es nicht mehr zwingend notwendig, Delegaten-Methoden innerhalb einer Klasse zu erzeugen.

```
namespace Lambda01
{
    delegate int myDelegate(int i, int j);

    public class Program
    {
        public static void Main(string[] args)
        {
            myDelegate add = delegate(int i, int j)
```

[1] Siehe Kapitel 18.

```
            {
                return i + j;
            };

            Console.WriteLine("{0}", add(3, 4).ToString());

            Console.ReadLine();
        }
    }
}
```

Listing 45.5 Zeiger auf Code über anonyme Methoden (aus dem Projekt Lambda01)

Delegaten können aber nicht nur Argumente für Methodenaufrufe liefern, sie können auch selbst als Argument übergeben werden.

```
namespace Lambda01
{
    public delegate int myDelegate(int i, int j);

    public class Program
    {
        public static void Main(string[] args)
        {
            myDelegate add = delegate(int i, int j)
            {
                return i + j;
            };

            Exec(add);

            Console.ReadLine();
        }

        public static void Exec(myDelegate del)
        {
            Console.WriteLine("{0}", del(3, 4).ToString());
        }

    }
}
```

Listing 45.6 Delegat als Parameter (aus dem Projekt Lambda01)

Einführung in Lambda-Ausdrücke

Mit der Einführung der Lambda-Ausdrücke in C# 3.5 wurde die Syntax aussprechender und einfacher. Listing 45.7 zeigt die Lambda-Version von Listing 45.5.

```
namespace Lambda01
{
    delegate int myDelegate(int i, int j);
```

```
    public class Program
    {
        public static void Main(string[] args)
        {
            myDelegate add = (int i, int j) => { return i + j; };

            Console.WriteLine("{0}", add(3, 4).ToString());

            Console.ReadLine();
        }
    }
}
```

Listing 45.7 Lambda-Ausdruck (aus dem Projekt Lambda01)

Durch die Delegaten-Definition `myDelegate(int i, int j)` ist klar, dass die zu übergebenden Parameter vom Typ `int` sind. Folglich ist es möglich, die Ausdruckssyntax durch Verzicht auf die redundante Typinformation zu vereinfachen.

```
myDelegate add = (i, j) => { return i + j; };
```

Befindet sich wie in Listing 45.7 im Ausdrucksrumpf nur eine `return`-Anweisung, so können sogar die geschweiften Klammern und das Schlüsselwort `return` im Ausdruck ebenfalls wegfallen.

```
myDelegate add = (i, j) => i + j;
```

Verwenden von Func-Delegaten-Typen

Mit dem Schlüsselwort `delegate` können Sie beliebige, eigene Delegaten definieren, die ggf. auch generisch sein können:[2]

```
public delegate T myDelegate<T>(T i, T j);

public class Program
{
    public static void Main(string[] args)
    {
        myDelegate<int> add = (i, j) => i + j;

        Console.WriteLine("{0}", add(3, 4).ToString());

        Console.ReadLine();
    }
}
```

[2] Siehe auch Kapitel 20, Abschnitt »Generische Delegaten und Prädikate«.

Seit Einführung von LINQ gibt es aber auch verschiedene vordefinierte Delegatentypen (Namespace `System.Linq`). Der Delegate `Func<>` ist vierfach überladen und unterstützt bis zu vier Parameter und einen Rückgabeparameter.

```
public delegate TR Func<TR>()
public delegate TR Func<T, TR>(T arg)
public delegate TR Func<T1, T2, TR>(T1 arg1,T2 arg2)
public delegate TR Func<T1, T2, T3, TR>(T1 arg1,T2 arg2,T3 arg3)
public delegate TR Func<T1, T2, T3, T4, TR>(T1 arg1,T2 arg2,T3 arg3,T4 arg4)
```

`TR` definiert den Rückgabetyp und steht in der generischen Typenliste immer an letzter Stelle. `T1` bis `T4` stellen die unterschiedlichen Parametertypen dar. Mit der Überladung für einen Parameter kann man das obige Code-Beispiel wie folgt umschreiben:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lambda02
{
    public delegate T Func<T, TResult>(T i, T j);

    public class Program
    {
        public static void Main(string[] args)
        {
            Func<int, int> add = (i, j) => i + j;

            Console.WriteLine("{0}", add(3, 4).ToString());

            Console.ReadLine();
        }
    }
}
```

Listing 45.8 Der Delegatentyp Func<> (aus dem Projekt Lambda02)

Prädikate und Projektionen

Es gibt grundsätzlich zwei Arten von Lambda-Ausdrücken: Prädikate[3] und Projektionen. Ein Prädikat liefert immer `true` oder `false` zurück und drückt aus, dass ein boolescher Ausdruck eine Bedingung erfüllt oder nicht. Bei einer Projektion handelt es sich um einen Ausdruckstyp, der sich von den Parametertypen unterscheiden kann. Listing 45.9 zeigt ein Beispiel für ein Prädikat.

[3] Siehe auch Kapitel 20, Abschnitt »Generische Delegaten und Prädikate« behandelt.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lambda03
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<string> customers = new List<string>();
            customers.Add("Shinja");
            customers.Add("Aaron");
            customers.Add("Angela");
            customers.Add("Dirk");

            string result = customers.Find( customer => customer.Equals("Angela"));
            List<string> results = customers.FindAll(c => c.Length > 5);

            Console.WriteLine("Ergebnis: {0}", result);

            foreach (string item in results)
            {
                Console.WriteLine("Ergebnis: {0}", item);
            }

            Console.ReadLine();
        }
    }
}
```

Listing 45.9 Prädikate eines Lambda-Ausdrucks (aus dem Projekt Lambda03)

Listing 45.10 zeigt noch einmal beide Arten von Lambda-Ausdrücken, wobei das Prädikat als Filter verwendet wird, während die Projektionen, wie z. B. `s.ToUpper()`, noch zusätzliche Ausdrücke bzw. Operationen ausführen.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Lambda03
{
    public class Program
    {
        //Func-Delegate
        static Func<string, bool> filter = s => s.Length == 5;  //Prädikat
        static Func<string, string> extract = s => s;           //Projektion
        static Func<string, string> project = s => s.ToUpper(); //Projektion

        public static void Main(string[] args)
```

```
        {
            List<string> customers = new List<string>();
            customers.Add("Shinja");
            customers.Add("Aaron");
            customers.Add("Angela");
            customers.Add("Dirk");

            if (filter(customers[1]))
                Console.WriteLine(customers[1].Length.ToString());

            Console.WriteLine(extract(customers[0]));

            Console.WriteLine(project(customers[2]));

            Console.ReadLine();
        }
    }
}
```

Listing 45.10 Projektion eines Lambda-Ausdrucks (aus dem Projekt Lambda04)

Um noch einmal die Verbindung zu den anonymen Methoden hervorzuheben: statt der vordefinierten `Func`-Delegaten könnte man auch folgende Definitionen verwenden:

```
static Func<string, bool> filter = delegate(string s)
{
   return s.Length == 5;
};
static Func<string, string> extract = delegate(string s)
{
   return s;
};
static Func<string, string> project = delegate(string s)
{
   return s.ToUpper();
};
```

Ausdrucksbäume

Es ist möglich, einen Lambda-Ausdruck als einen Ausdrucksbaum zu repräsentieren und ggf. zu manipulieren. Ein Ausdrucksbaum stellt somit eine Repräsentation eines Lambda-Ausdrucks dar. Um einen Lambda-Ausdruck in einen Ausdrucksbaum umwandeln zu können, benötigt man den Namespace `System.Linq.Expressions` mit dem generischen Typ `Expression<T>`.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

using System.Linq.Expressions;

namespace Lambda05
```

```
{
    public class Program
    {
        public static void Main(string[] args)
        {
            int val = 4;

            Expression<Func<int, bool>> exp = p => p < 7;

            if (exp.Compile()(val))
                Console.WriteLine("smaller than 7");
            else
                Console.WriteLine("greater than 7");

            BinaryExpression body = (BinaryExpression) exp.Body;
            ParameterExpression left = (ParameterExpression) body.Left; // p
            ConstantExpression right = (ConstantExpression) body.Right; // 7
            Console.WriteLine("{0} {1} {2}", left.Name, body.NodeType, right.Value);

            Console.ReadLine();
        }
    }
}
```

Listing 45.11 Auslesen eines Ausdruckbaumes (aus dem Projekt Lambda05)

Listing 45.11 definiert einen Ausdruck für: `p` kleiner `7`. Sie können einen solchen Lambda-Ausdruck, wie im Listing gezeigt, durch Aufruf der Methode `Compile` ausführen lassen oder in seinen Ausdrucksbaum zerlegen.

HINWEIS Die Elemente des Ausdrucksbaums können analysiert, aber nicht verändert werden.

Umgekehrt ist es auch möglich, einen Lambda-Ausdruck zur Laufzeit aus einem Ausdrucksbaum zu erzeugen:

```
int val = 4;

ParameterExpression parameterExpression = Expression.Parameter(typeof(int), "n");
ConstantExpression constExpression = Expression.Constant(7, typeof(int));
BinaryExpression lessThan = Expression.LessThan(parameterExpression, constExpression);
Expression<Func<int, bool>> lambdaExpression = Expression.Lambda<Func<int, bool>>(
                                                    lessThan,
                                                    new ParameterExpression[] { parameterExpression });

if (lambdaExpression.Compile()(val))
    Console.WriteLine("smaller than 7");
else
    Console.WriteLine("greater than 7");
```

Listing 45.12 Lambda-Ausdruck durch Ausdrucksbaum erzeugen (aus dem Projekt Lambda05)

Hier wird die Funktionsweise des Codes aus Listing 45.11 durch unterschiedliche Ausdrucksdefinitionen (`ParameterExpression`, `ConstantExpression` und `BinaryExpression`) zur Laufzeit nachgebaut. Über den generischen Typ `Expression<T>` wird der Lambda-Ausdruck in `lambdaExpression` erzeugt und über die Methode `Compile()` zur Ausführung gebracht. Tabelle 45.1 listet die wichtigsten `Expression`-Klassen zur Erzeugung eines Ausdruckbaumes auf.

Typ	Beschreibung
`Expression`	Abstrakte Basisklasse aller Ausdrucksklassen.
`Expression<T>`	Generische Klasse, die einen Ausdrucksbaum verwaltet und den Lambda-Ausdruck binär darstellt.
`BinaryExpression`	`BinaryExpression` wird für Ausdrücke mit zwei Operanden verwendet, die die Eigenschaftswerte `Left` und `Right` besitzen.
`ConditionalExpression`	Mithilfe der Klasse `ConditionalExpression` wird eine Bedingung in einem Ausdrucksbaum definiert.
`ConstantExpression`	Die Klasse `ConstantExpression` erlaubt, einen konstanten Wert zu setzen, der in einem Ausdruck verwendet wird.
`LambdaExpression`	Diese Klasse stellt einen Lambda-Ausdruck dar, der ausgeführt werden kann.
`MethodCallExpression`	`MethodCallExpression` wird für Ausdrücke verwendet, die einen Methodenaufruf darstellen.
`ParameterExpression`	Diese Klasse definiert die Parameter, die in einem Ausdruck verwendet werden.
`UnaryExpression`	`UnaryExpression` definiert einen unären Ausdruck, der in einem Lambda-Ausdruck verwendet wird.
`TypeBinaryExpression`	Diese Klasse definiert eine Operation zwischen einem Ausdruck und einem Typen.

Tabelle 45.1 Die wichtigsten Klassen zur Erzeugung bzw. Darstellung von Ausdrucksbäumen