15.–18. 09. 2008
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Noch Schärfer!

Neues in C# 3.0

Oliver Szymanski

MATHEMA Software GmbH

Inhalt

- Implicitly typed local variables
- Extension methods
- Lambda expressions
- Object initializers
- Collection initializers
- Anonymous Types
- Implicitly typed arrays
- Query expression

C# 3.0

Implicitly typed local variables

- Implizite Bestimmung eines Variablentyps
- Bereits bei der Deklaration möglich

```
var i = 3;
var str = "Hello World";
var d = 1.0;
var numbers = new int[] {1, 2, 3};
var orders = new Dictionary<int,Order>();
```

- Bitte nicht benutzen ...
  - Außer vielleicht, wenn es wirklich Sinn macht
    - Wirklich Sinn!

Implicitly typed local variables

~~var e;~~
~~var z = null;~~
~~var array = {1 , 2, 3};~~

```
int[] numbers = { 1, 3, 5, 7, 9 };

foreach (var n in numbers)
        Console.WriteLine(n);
```

Extension Methods

- Statische Methoden, die als Instanzmethoden ausgeführt werden können
- Erweitern bestehende Typen
- Extension methods werden in statischen Klassen definiert
- Erster Parameter bekommt das Schlüsselwort „this“, der danach angegebene Typ wird erweitert

Extension Methods

- Schlüsselwort „this“ definiert Extension Method
- Klasse muss statisch sein

```
namespace Mathema.Examples {
    public static class Extensions {
        public static int ToInt32(this string s) {
            return Int32.Parse(s);
        }
    }
}
```

- Klasse ist Methodensammlung für zu erweiternde Typen

Extension Methods

- Extension methods können statisch über die statische Klasse aufgerufen werden

```
Extension.ToInt32("123");
```

- Oder als Instanzmethode des erweiterten Typ

```
using Mathema.Examples;
...
string s = "123";
s.ToInt32();
```

- Als Instanzmethode entfällt der erste Parameter und die statische Klasse muss importiert worden sein

Extension Methods - Anmerkungen

- Extension Methods nur in Ausnahmefällen nutzen
- Attribut [System.Runtime.CompilerServices.Extension] =>
  - Schlüsselwort „this“ als erster Parameter ist Sprachfeature
- Auflösung erfolgt bei Kompilierung

- Extension-Sammlung für Zweckgebundene Methoden
  - importiert werden wenn benötigt
  - z.B. mit System.Query für Query Methoden bei Collections
- Alle diese Methoden können einfach ausgetauscht werden

- Auf Innereien kommt man natürlich nicht!

Lambda Expressions

- L.A. sind vergleichbar mit anonymen Methoden
- L.A. bieten funktionale Schreibweise
- Bestehen Eingabeparametern und Statements/Ausdrücken

```
x => x+1
x => {return x + 1;}
(int x) => x+1
(x, y) => x*y
```

- Eingabeparametertypen können implizit angegeben werden

Lambda Expressions

- Lambda expressions können in Delegates umgewandelt werden
  - überall wo ein Delegate erwartet wird, kann man eine Lambda Expression übergeben

```
delegate R Func<A,R>(A arg);

...

Func<int,int> f1 = x => x + 1; // Ok
Func<int,double> f2 = x => x + 1; // Ok
Func<double,int> f3 = x => x + 1; // Error
```

Lambda Expressions – Beispiel 1

```
static Z calc <X,Y,Z> (X value,
                       Func<X,Y> f1,
                       Func<Y,Z> f2)
{
    return f2(f1(value));
}

double seconds = calc ("1:15:30",
                       s => TimeSpan.Parse(s),
                       t => t.TotalSeconds);
```

Lambda Expressions – Beispiel 2

```
namespace System.Query {
    public static class Sequence {
        public static IEnumerable<S> Select<T,S>(
            this IEnumerable<T> source,
            Func<T,S> selector)
        {
            foreach (T element in source) yield
                return selector(element);
        }
    }
}
...

List<Customer> c = ...;
IEnumerable<string> n = c.Select(c => c.Name);
```

Expression Trees

- Lambda expressions können in expression trees umgewandelt werden

```
Expression<Func<int, bool>> filter = n => n < 5;

BinaryExpression    body  = (BinaryExpression)filter.Body;
ParameterExpression left  = (ParameterExpression)body.Left;
ConstantExpression  right = (ConstantExpression)body.Right;

Console.WriteLine("{0} {1} {2}",
                  left.Name, body.NodeType, right.Value);
```

- Dies führt zu der Ausgabe: n LT 5

Object Initializers

- Verkürzte Schreibweise für Erzeugen und Initialisieren

```
public class Point {
    int x, y;
    public int X { get { return x; } set { x = value; } }
    public int Y { get { return y; } set { y = value; } }
}

var a = new Point { X = 0, Y = 1 };

statt
var a = new Point();
a.X = 0;
a.Y = 1;
```

Object Initializers - Mehr^^

- Verkürzte Schreibweise für Erzeugen und Initialisieren

```
public class Rectangle {
    Point p1, p2;
    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}

var r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

Object Initializers – noch komfortabler

- Verkürzte Schreibweise für Erzeugen und Initialisieren

```
public class Rectangle {
    Point p1 = new Point();
    Point p2 = new Point();
    public Point P1 { get { return p1; } set { p1 = value; } }
    public Point P2 { get { return p2; } set { p2 = value; } }
}

var r = new Rectangle {
    P1 = new Point { X = 0, Y = 1 },
    P2 = new Point { X = 2, Y = 3 }
};
```

Collection Initializers

- Beliebige Collections mit vereinfachter Schreibweise

```
List<int> digits =
    new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Collection Initializers – Mehr^^

```
public class Contact {
    string n;
    List<string> pn = new List<string>();
    public string Name { get { return n; } set { n = value; } }
    public List<string> PhoneNumbers { get { return pn; } }
}

var contacts = new List<Contact> {
    new Contact {
        Name = "Chris Smith",
        PhoneNumbers = { "206-555-0101", "425-882-8080" }
    },
};
```

Anonyme Typen

- „new“ um Objekte eines anonymen Typen anzulegen

```
Var p = new {Name="Pete", Id="3"}
```

- Anonymer Typ wird dann implizit definiert

```
class _anonym1 {
    string n;
    string id;

    public string Name   {get{return n;}  set{n=value}}
    public string Id     {get{return id;} set{id=value}}
}
```

Anonyme Typen

- Zwei Objekte von anonymen Typen mit gleichen Properties (Name und Typ) und gleicher Property-Reihenfolge sind kompatibel

```
Var p = new {Name="Pete",    Id="3"}
Var z = new {Name="Ted",     Id="1"}
p = z; // da Typkompatibel
```

Implicitly Types Arrays

- Wir legen Arrays jetzt auch implizit typisiert an
  - wer's unbedingt möchte ...

```
var a = new[] { 1, 10, 100, 1000 }; // int[]
var b = new[] { 1, 1.5, 2, 2.5 }; // double[]
var c = new[] { "hello", null, "world” }; // string[]
var d = new[] { 1, "one", 2, "two" };

var contacts = new[] {
    new {
        Name = "Chris Smith",
        PhoneNumbers = new[] { "206-555-0101", "425-882-8080" }
    }, new {
        Name = "Bob Harris",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

LINQ – Query Expressions

- Es gibt eine eingebaute Query Language
- SQL-ähnliche Query zu Datentypen/Collections/Arrays
- Mit z.B. LINQ2SQL Anbindung an DBs

```
string[] names = { "Burke", "Connor",
                   "Harris", "David" };

IEnumerable<string> expr =
                    from s in names
                    where s.Length == 5
                    orderby s
                    select s.ToUpper();
```

LINQ – Query Expressions

- Sortierung, Filtern, Joins, Gruppierung möglich
- Explizite Typangaben möglich

```
from Customer c in customers
where c.City == "London"
select c
```

LINQ – Query Expressions

- Umwandlung in Methodenaufrufe geschieht implizit beim Kompilieren

```
from Customer c in customers
where c.City.Equals("London")
select c

entspricht

customers.Cast<Customer>().
                Where(c => c.City.Equals("London"))
```

LINQ – Query Expressions

- Umwandlung in Methodenaufrufe geschieht implizit beim Kompilieren

```
from c in customers
join o in orders on c.CustomerID equals
o.CustomerID
select new { c.Name, o.OrderDate, o.Total }

entspricht

customers.Join(   orders,
                  c => c.CustomerID,
                  o => o.CustomerID,
                  (c, o) =>  new { c.Name,
                                   o.OrderDate, o.Total })
```

LINQ – Query Expressions

- Als Standard-Query Methoden über Extension Methods definiert
- Müssen für die Nutzung importiert werden
- Für eigene Datentypen entsprechende Methoden auch implementieren
  - um Query expressions zu unterstützen (als Instanz- oder Extension- Methoden)

LINQ – Query Expressions

- Beispiel: „Where“

```
using System.Query; // Operatoren importieren
...

IEnumerable<string> expr =
    Sequence.Where(names, s => s.Length < 6);
```

LINQ – Query Expressions

- Beispiel: „Where“

```
namespace System.Query {

  public static class Sequence {
    public static IEnumerable<T> Where<T>(
                this IEnumerable<T> source,
                Func<T, bool> predicate) {

      foreach (T item in source)
        if (predicate(item))
          yield return item;
    }
  }
}
```

.NET Framework 3.5

Beispiele & Fragen

Vielen Dank!

Oliver Szymanski

MATHEMA Software GmbH

Firma

- Hier können Sie, wenn Sie möchten, Informationen über Ihre Firma einfügen