BASTA!

.NET, VISUAL STUDIO & MORE

Ian Cooper & Oliver Sturm

Alles, was Sie über C# 3.0 wissen müssen

Power Workshop

Microsoft Most Valuable Professional

Overall Agenda

- New language features in C# 3.0
- Cool things to do with the new features
- LINQ
- Functional Programming with C# 3.0 – as much as we have time for

New language features in C# 3.0

Ian Cooper & Oliver Sturm

Agenda

- Das “var”-Keyword
- Extension Methods
- Auto-Implemented Properties
- Initializers
- Anonymous Types
- Partial Methods
- Lambda Expressions
- Expression Trees
- Query Expressions

Das “var”-Keyword

```
int i = 5;
string s = "Hello";
double d = 1.0;
int[] numbers = new int[] {1, 2, 3};
Dictionary<int,Order> orders =
   new Dictionary<int,Order>();
```

Das “var”-Keyword

```
var i = 5;
var     s = "Hello";
var     d = 1.0;
var    numbers = new int[] {1, 2, 3};
var                    orders =
    new Dictionary<int,Order>();
```

GLEICHES RESULTAT

Das “var”-Keyword

Demo

Extension Methods

```
namespace MyStuff {
    public static class Extensions {
        public static string Concatenate(
            this IEnumerable<string> strings,
            string separator)
        { … }
    }
}
```

Extension method

```
using MyStuff;
```

Macht Extension verfügbar

```
string[] names = new string[] {"Mary", "Mungo", "Midge"};
string s = names.Concatenate(", ");
```

Extension Methods

Demo

Auto-implemented Properties

```
private string stringValue;
public string StringValue {
   get { return stringValue; }
   set {
      stringValue = value;
   }
}
```

Auto-implemented Properties

```
public string StringValue {
   get;
   set;
}
```

GLEICHES RESULTAT

Auto-implemented Properties

Demo

Initializers

```
class Person {
    public Person( ) { }
    public Person(string name, int age) {
        this.Name = name;
        this.Age = age;
    }
    public string Name { get; set; }
    public int Age { get; set; }
}
```

Constructor initialization:

```
var person = new Person("Willy", 33);
```

Initializers

Demo

Anonymous Types

```
var person = new Person {
                Name = "Willy",
                Age = 33
             };
```

Anonymous Types

Demo

Partial Methods

- Nur in partial classes
- Aufrufe werden “wegoptimiert”, wenn die Methode nicht implementiert ist
- Performante Alternative zu Events, für Codegeneratoren

Partial Methods

Demo

Lambda Expressions

```
delegate int
  TakeIntReturnIntDelegate(int input);
...
int Add10(int input) {
      return input + 10;
}
...
void DoSomething(
  TakeIntReturnIntDelegate function) {
      Console.WriteLine(function(10));
}
...
DoSomething(Add10);
```

Lambda Expressions

```
delegate int
  TakeIntReturnIntDelegate(int input);

int Add10(int input) {
      return input + 10;
}
...
void DoSomething(
  TakeIntReturnIntDelegate function) {
      Console.WriteLine(function(10));
}
...
DoSomething(Add10);
```

Lambda Expressions

```
delegate int
  TakeIntReturnIntDelegate(int input);
...
void DoSomething(
  TakeIntReturnIntDeleg
      Console.WriteLine
}
...
DoSomething(Add10);
```

```
int Add10(int input) {
      return input + 10;
}
```

```
Delegate(

            );
```

Lambda Expressions

```
delegate int
  TakeIntReturnIntDelegate(int input);
...
void DoSomething(
  TakeIntReturnIntDelegate function) {
      Console.WriteLine(function(10));
}
...
DoSomething( delegate(int input){
                  return input + 10;
             });
```

Lambda Expressions

```
void DoSomething(
  Func<int, int>                function) {
      Console.WriteLine(function(10));
}
...
DoSomething(input => input + 10);
```

Lambda Expressions

Demo

Expression Trees

Speicherung in einer Variablen

```
class Program {
    d Main() {
    double> f = x => x * 2 + 1;
    Line(f(10));
```

Speicherung in einer Liste

```
var funcs = new List<Func<double, double>>();
funcs.Add(x => x * 2 - 1);
funcs.Add(x => 1 / x);
funcs.Add(x => Math.Sqrt(x));

foreach (var f in funcs) Console.WriteLine(f(10));
```

Exp

Expression und Expression<T> speichern Lambdas in einer Form, die Analyse und synthetische Erzeugung zulässt.

Ein anderer Datentyp erlaubt die Speicherung von Lambdas als Daten

```
publi
Expre
```

```
tomer), "c");

y("State")),
    Expression.Constant("NY"));
Expression<Predicate<Customer>> test =
  Expression.Lambda<Predicate<Customer>>(expr,c);
```

Expression Trees

Demo

Query Expressions

```
from c in customers
where c.State == "NY"
select new { c.Name, c.Phone };
```

Spezielle Syntax wird vom Compiler umgewandelt

```
customers
.Where(c => c.State == "NY")
.Select(c => new { c.Name, c.Phone });
```

Query Expressions

Demo

C# 3.0 new language features Summary

- Neue Sprachfeatures machen C# 3.0-Code einfacher und kürzer
- Viele der Features sind “Compilertricks” und brauchen keine Framework-Unterstützung (auch verwendbar mit .NET 2!)
- C# bewegt sich in eine funktionale Richtung

Cool things to do with the new C# 3.0 features

Ian Cooper & Oliver Sturm

Agenda

- Control.Invoke
- Ruby-style iterations
- Dynamic querying
- Ruby-style ranges

Control.Invoke

- Control.Invoke wird benötigt, um Code im UI-Thread ausführen zu lassen (Windows Forms)
- Mit Hilfe von C# 3.0-Features kann man das syntaktisch viel einfacher gestalten

Control.Invoke

Demo

Ruby-style iterations

Demo

Dynamic querying

- Expressions können dynamisch (zur Laufzeit) erzeugt werden
- Mechanismen wie QBE (Query By Example) lassen sich so sehr einfach verwirklichen

Dynamic querying

Demo

Ruby-style ranges

Demo

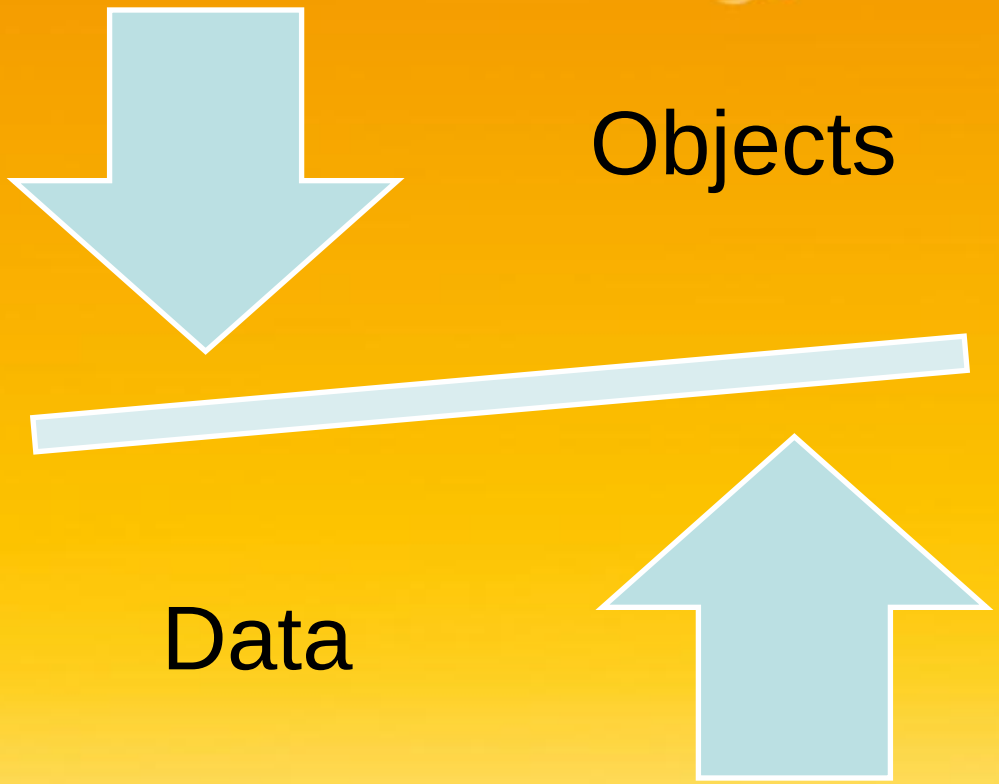
Cool things you can do - summary

- Obwohl die neuen Sprachfeatures hauptsächlich für LINQ eingeführt wurden, sind sie vielfältig verwendbar.

LINQ – change the way you write queries forever

Ian Cooper

Lost in translation



Ian Cooper & Oliver Sturm

Imperative

```
string retval = "";
foreach (Foo foo in mylist)
{
 string desc =
    foo.description();
    if (desc != "")
    {
        if (retval != "")
            retval += "\n";
        retval += desc;
     }
 }
```

Declarative

```
concat $ List.intersperse "\n" $
filter (/= "") $ map description
mylist
```

The LINQ Project

C# 3.0

VB 9.0

Others…

.NET Language Integrated Query

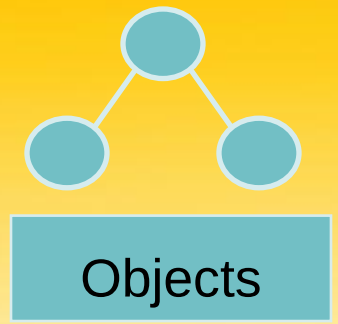
LINQ to Objects

LINQ to DataSets

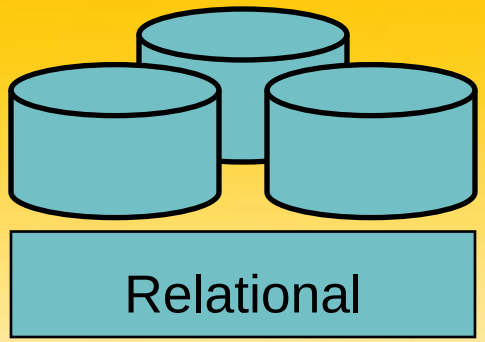
LINQ to SQL

LINQ to Entities

LINQ to XML



Objects



Relational

```
<book>
   <title/>
   <author/>
   <year/>
   <price/>
</book>
```

XML

Ian Cooper & Oliver Sturm

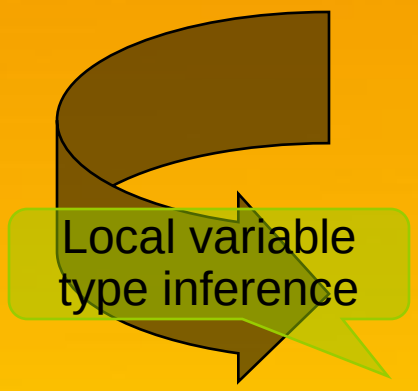
LINQ to Objects

Demo

C# 3.0 Language Extensions

Query expressions

```
var contacts =
    from c in customers
    where c.State == "WA"
    select new { c.Name, c.Phone };
```



Local variable type inference

Lambda expressions

```
var contacts =
    customers
    .Where(c => c.State == "WA")
    .Select(c => new { c.Name, c.Phone });
```

Extension methods

Anonymous types

Object initializers

Ian Cooper & Oliver Sturm

Query Methods

Restriction	Where
Projection	Select, SelectMany
Ordering	OrderBy, ThenBy
Grouping	GroupBy
Joins	Join, GroupJoin
Quantifiers	Any, All
Partitioning	Take, Skip, TakeWhile, SkipWhile
Sets	Distinct, Union, Intersect, Except
Elements	First, Last, Single, ElementAt
Aggregation	Count, Sum, Min, Max, Average
Conversion	ToArray, ToList, ToDictionary
Casting	OfType<T>, Cast<T>

ADO.NET

```
using(SqlConnection c = new SqlConnection(…)
{
        c.Open();
        SqlCommand cmd = new SqlCommand(
          @"SELECT c.Name, c.Phone
              FROM Customers c
              WHERE c.City = @p0");
        cmd.Parameters.AddWithValue("@p0", "London“);
        using(DataReader dr = c.Execute(cmd)
        {
              while (dr.Read())
              {
                  string name = dr.GetString(0);
                  string phone = dr.GetString(1);
                  DateTime date = dr.GetDateTime(2);
              }
        }
}
```

Queries in quotes

Loosely bound arguments

Loosely typed result sets

No compile time checks

LINQ to SQL

```
public class Customer { … }

public class Northwind : DataContext
{
    public Table<Customer> Customers;
    …
}
```

Classes describe data

Tables are like collections

Strongly typed connections

```
Northwind db = new Northwind(…);
var contacts =
    from c in db.Customers
    where c.City == "London"
    select new { c.Name, c.Phone };
```

Integrated query syntax

Strongly typed results

LINQ to SQL

Demo

System.XML

```
XmlDocument doc = new XmlDocument();
XmlElement contacts = doc.CreateElement("contacts");
foreach (Customer c in customers)
    if (c.Country == "USA") {
        XmlElement e = doc.CreateElement("contact");
        XmlElement name = doc.CreateElement("name");
        name.InnerText = c.CompanyName;
        e.AppendChild(name);
        XmlElement phone = doc.CreateElement("phone");
        phone.InnerText = c.Phone;
        e.AppendChild(phone);
        contacts.AppendChild(e);
    }
doc.AppendChild(contacts);
```

```
<contacts>
  <contact>
    <name>Great Lakes Food</name>
    <phone>(503) 555-7123</phone>
  </contact>
  …
</contacts>
```

Imperative model

Document centric

No integrated queries

Memory intensive

LINQ to XML

```
XElement contacts = new XElement("contacts",
    from c in customers
    where c.Country == "USA"
    select new XElement("contact",
        new XElement("name", c.CompanyName),
        new XElement("phone", c.Phone)
    )
);
```

- Declarative model
- Element centric
- Integrated queries
- Smaller and faster

LINQ to XML

Demo

LINQ Summary

- New approach to querying data.
- Declarative as opposed to imperative approach.
- One common approach vs. Many.
- Type safe

Functional Programming in C# 3.0

Oliver Sturm

Agenda

Fokus: Was C# alles enthält in Zusammenhang mit FP und wie es funktioniert

- Was ist Funktionale Programmierung?
- FP Features in C# 3.0 und .NET 3.5
- Map, Filter und Reduce
- Currying, Partial Application und Composition
- Was hat ein C#-Programmierer von FP?

Was ist Funktionale Programmierung?

- Ein Programmiermodell
- Fokus auf der Anwendung von Funktionen
- Vermeidung von Zustandsspeicherung und veränderbaren Daten
- Bekannte Sprachen sind u.a. Lisp, Scheme, Haskell, ML and (neuerdings) F#
- FP-Sprachen haben gewöhnlich Features zur Unterstützung von support Higher Order Functions, Currying, Rekursion, List Comprehensions, …
- Viele imperative und objektorientierte Sprachen haben heute FP-Features

Warum ist FP interessant?

- Fördert Modularisierung
- Lazy evaluation → Höhere Effizienz
- Das Ziel, Nebeneffekte zu vermeiden, hat mehrere Vorteile: Skalierung, Optimierung, Debugging, Testing
- C# 3.0 unterstützt viele wichtige FP-Techniken

Was .NET enthält

Demo

Intermezzo – Map, Filter und Reduce

- **Map/Select** tut etwas mit jedem Element einer Liste
- **Filter/Where** extrahiert Elemente aus einer Liste, basierend auf Konditionen
- **Reduce/Aggregate** fasst Elemente in einer Liste zusammen, mithilfe einer Berechnung
- Select, Where und Aggregate sind.NET 3.5-Implementationen dieser Funktionen

Was .NET enthält - Fortsetzung

Demo

Map, Filter, Reduce

- Map tut etwas mit jedem Element einer Liste
- Filter extrahiert Elemente aus einer Liste, basierend auf Konditionen
- Reduce fasst Elemente in einer Liste zusammen, mithilfe einer Berechnung

Map, Filter, Reduce

Demo

Currying und Partial Application

- Currying: Eine Funktion, die mehrere Parameter entgegennimmt, in eine Kette von Funktionen konvertieren, die jeweils einen Parameter entgegennehmen und die nächste Funktion zurückliefern, bis die letzte Funktion der Kette die Berechnung mit allen Parametern ausführen und das Resultat zurückliefern kann.
- Partial Application: Einen oder mehrere Parameter einer Funktion in ge-”curry”-tem Format festlegen, so dass eine neue Funktion mit einem spezielleren Einsatzgebiet entsteht.

Manuelles und Automatisches Currying

Demo

Function Construction

- Die Idee, neue Funktionen aus vorhandenen zu erzeugen
- Fördert Modularisierung auf der Ebene von Funktionen
- Partial Application ist eine Möglichkeit
- Composition ist eine andere:

  Angenommen `B = f1(A), C = f2(B)`

  → `C = f2(f1(A))`

Composition

Demo

Ansätze kombinieren

- Ziel: Erzeugung einer Funktion `int sumOfOddNumbers(int)`, basierend auf Reduce
- Partial Application anwenden, um eine Berechnungsstrategie für Reduce zu definieren, und einen Algorithmus für Sequenzerzeugung
- Composition anwenden zur Vereinfachung der Benutzung

Function Construction

Demo

FP in C# - was sind die Vorteile?

- An funktionale Modularisierung muss man sich erst gewöhnen, aber es lohnt sich
- Unit testing kann von der Philosophie der Vermeidung von Nebeneffekten profitieren
- Skalierung ist einfacher, ob Sie Ihre eigenen Threads verwenden, Threadpools oder ParallelFX
- Man erledigt einfach mehr – versuchen sie es mal!
- ABER: Stellen Sie sicher, dass Ihre Kollegen die ganze Sache auch verstehen!

FP – Zusammenfassung

- C# hat gute Unterstützung für die Ideen der Funktionalen Programmierung
- Etwas Handarbeit muss noch getan werden
- Die Syntax ist manchmal etwas sonderbar
- FP bietet “Glueing”-Techniken (Currying, Partial Application, Composition) auf einer funktionalen Ebene und bietet damit einen neuen Ansatz der Modularisierung

Vielen Dank

Feel free to contact us at any time about the content of the workshop

ian_hammond_cooper@yahoo.co.uk

oliver@sturmnet.org