

Linq and C# 3.0

A new way and language to query data

Overview

- What is Linq?

- Flavors of Linq

- Linq to XML
- Linq to DataSets
- Linq to SQL

- Linq under the covers

- Linq deferred

- Q&A and/or Discussion

Introducing...

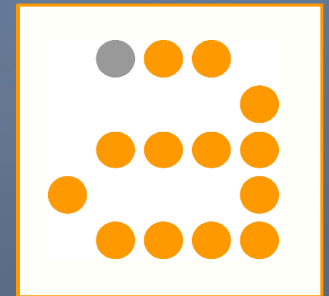
Alex Thissen

- Trainer/coach
- Weblog at <http://www.alexthissen.nl>



INETA

- Next step in user group evolution
- By and for user group community



Class-A

- Knowledge provider
- Training and coaching on Microsoft development
- www.class-a.nl

What is Linq?

- Linq stands for Language Integrated Query
- New set of keywords in C# 3.0 and VB.NET 9.0 to express queries over data

```
Dim homeMatchesWon As IEnumerable(Of Match) =  
    From m In matches  
    Where m.GoalsHome > m.GoalsAway  
    Order By m.HomeTeam, m.AwayTeam Descending  
    Select m
```

Linq project

Languages

C# 3.0

VB 9.0

Other

Linq

Standard Query Operators

Linq to
Objects

Linq to
XML

Linq to
SQL

Linq to
DataSets

Linq to
Entities

Query expressions

- Queries expressed with language keywords
- Special syntax

```
from id in source
[ join id in source on expr equals expr ]
[ let id = expr ]
[ where expr ]
[ orderby ordering, ordering, ... ]
select expr | group expr by key
[ into id ]
```

- Query expressions are translated into invocations of methods

Standard Query Operators

- Known set of methods to express queries
- Larger set than is integrated in language
- An other programming model for queries
 - Also called explicit dot notation
- Standard Query Operators are provided by extension methods
 - on any object that implements `IEnumerable<T>`

Explicit dot notation example

```
IEnumerable<Team> winningHomeTeams =  
    matches  
        .Where(m => m.GoalsHome > m.GoalsAway)  
        .OrderBy(m => m.HomeTeam.Name)  
        .Select(m => m.HomeTeam);
```

Standard Query Operators

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

Demo: Linq fundamentals

- Linq queries
- Standard query operators
- Explicit dot notation

Linq to XML

Queries with hierarchical data

Linq to XML

- Power of Linq brought to data in XML format
 1. Perform queries over XML data
 2. New API to manipulate XML
 - Alternative to XML DOM API
 3. Create XML data with query expressions

New API for XML

- Builds on existing knowledge of DOM
- Element centric instead of document centric
- Functional construction of XML
- New classes represent various parts of XML
 - For example:
`XElement`, `XAttribute`, `XDocument`,
`XText`
- Base class for all node types: `XNode`

Extra Linq to XML query operators

- Defined as extension methods on

`IEnumerable<XElement>` in `XElementSequence`

- Operators for XPath axis:

- `Ancestors`, `SelfAndAncestors`,
`Descendants`, `SelfAndDescendants`,
`ElementsBefore/AfterThis`,
`NodesBefore/AfterThis`

- Operators for node sets:

- `Nodes`, `Elements`, `Attributes`
- XPath axis sets: `Nodes`, `DescendantNodes`,
`AncestorNodes`, `SelfAndDescendantNodes`

More Linq to XML

- Add annotations to the `XContainer` nodes (`XElement` and `XDocument`)
 - Do not show up in XML output
- `XStreamingElement` defers generation of XML element content
 - Use in place of `XElement`
 - `Save` method triggers creation
 - Lazy/streaming output of XML to file or writer

Demo: Linq to XML

- Querying hierarchical data
- Creating XML
- Using Linq to XML API

Linq to DataSets

The missing functions of ADO.NET

Linq over DataSets

- Enables querying over `ADO.NET DataTable`
- `System.Data.DataTable` is central class
- Adds some helpful extension methods to easily load data into a `DataTable`
 - `LoadSequence`:
Loads data into `DataTable`
 - `ToDataTable`:
Convert any `IEnumerable<T>` into a newly created `DataTable`

Extra Linq to DataSet query operators

- DataTable

- Work with sets of DataRow
- `DistinctRows`, `EqualAllRows`,
`ExceptRows`, `IntersectRows`, `UnionRows`

- DataRow

- Adds strong typing and null support
- `Field<T>` reads from fields
- `SetField<T>` sets values on fields

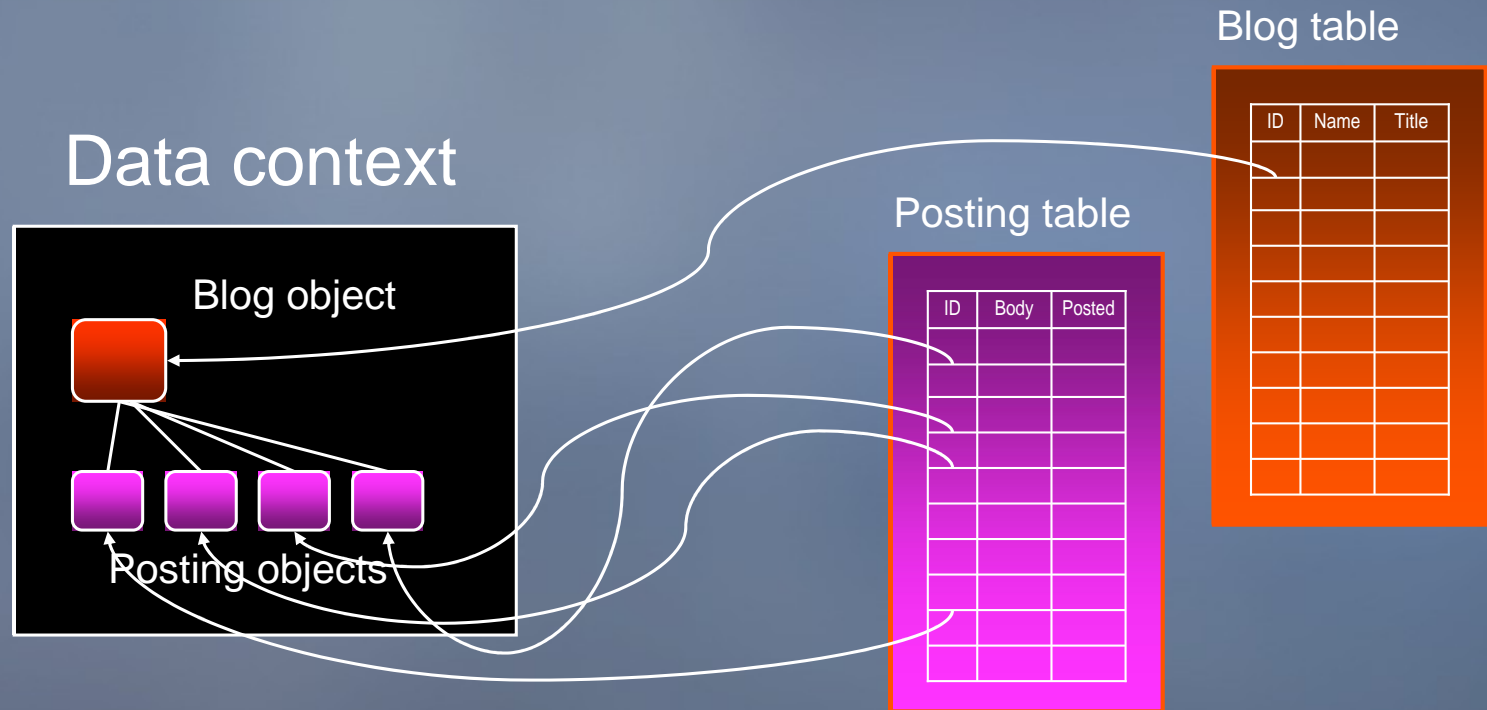
Linq to SQL

Queries and object/relational mapping

Linq to SQL for relational data

- Language integrated data access
- Mapping of CLR types to database tables
 - Object/Relation mapping technology
 - Translates Linq queries to SQL statements
- Builds on ADO.NET and .NET Transactions
- Persistence services
 - Automatic change tracking of objects
 - Updates through SQL statements or stored procedures

Mapping strategy



- Relationships map to collection properties
- 1 to 1 correspondence between type and table
- Single table inheritance is supported

Mapping from objects to relations

- Two mapping mechanisms between CLR and database world
 1. Attributes on CLR types
 2. XML mapping file
- Table<T> class handles mapping and materialization of objects into context
- SqlMetal.exe tool and DLinQ designer help in generation of both mapping types

Attribute-based mappings

```
[System.Data.DLinq.Table(Name="Blogs")]
public partial class Blog
{
    private string _Name;
    [System.Data.DLinq.Column(Name="Name",
        Storage="_Name",
        DbType="nvarchar NOT NULL")]
    public virtual string Name {
        get { return this._Name; }
        set {
            if ((this._Name != value)) {
                this.OnPropertyChanging("Name");
                this._Name = value;
                this.OnPropertyChanged("Name");
            }
        }
    }
}
```

Contexts of data objects

- Objects returned from Linq to SQL queries are materialized into DataContext
- DataContext class handles
 - Change tracking
 - Object identity
- Currently load context of objects can be saved, discarded or accepted
- EntitySet<T> uses lazy loading of related collections into context

Notifying changes

- UI and object graphs have to keep in sync with changes, inserts and deletes
- Achieved with implementations of interfaces:
 - `System.Data.Linq.INotifyPropertyChanging`
 - `System.ComponentModel.INotifyPropertyChanged`
- Allows collections and referenced objects to be kept up to date

Future of Linq to SQL

- Linq to SQL competes in O/R mapping space with ADO.NET Entities
- ADO.NET 3.0 will introduce Entities
 - Mapping from conceptual to logical model
 - More flexible and powerful mapping approach
- Will Linq to SQL survive?
 - Coexist with ADO.NET Entities?
 - Be dropped like ObjectSpaces was in 2005?

Linq under the covers

How Linq works

Working with sources of data

- Query operators work on sets of data
 - `from m in matches`
- Everything that implements `IEnumerable<T>` can be queried
- Implementing `IEnumerable<T>` seems to get a class new methods
- Past the syntactic sugar it is all about Standard Query Operators

Linq under the covers

Query expression

```
Match[] matches = MatchService.GetMatches();  
IEnumerable<Team> winningHomeTeams =  
    from m in matches  
    where m.GoalsHome > m.GoalsAway  
    order by m.HomeTeam.Name  
    select m.HomeTeam;
```

Compiler creates code similar to

```
IEnumerable<Team> winningHomeTeams =  
    matches  
        .Where(m => m.GoalsHome > m.GoalsAway)  
        .OrderBy(m => m.HomeTeam.Name)  
        .Select(m => m.HomeTeam);
```

C# 3.0: Extension methods

- Project instance methods onto existing classes
- Declared in static class with static methods
- Introducing another `this` keyword
 - Type of first argument defines class to extend
 - Compiler emits `[ExtensionAttribute]` to class and extension methods
- Import namespace to bring extension methods into scope
- Instance methods take precedence over static

Extension methods example

```
namespace System.Query
{
    public static class Sequence
    {
        public static IEnumerable<S> Select<T>(
            this IEnumerable<T> source,
            Func<T, S> selector) { ... }
        // ...
    }
}

// To use, import namespace of extension class
using System.Query;

IEnumerable<string> =
    blogs.Select(b => b.Name.Length > 0);
```

Back to IEnumerable<T>

- Standard Query Operators are available for all `IEnumerable<T>` implementing types
- Extension methods are in class `System.Query.Sequence`
- Implementation of `Sequence` relies heavily on iterators and `yield` keyword

Iterators

- Perform iterations in C# with `foreach`
- Iterator pattern by implementation of `IEnumerable` interface
- Lots of code required
 - `IEnumerable` must return object that implements `IEnumerator`
 - `Enumerator` class must explicitly maintain state of iteration

Yield

- C# 2.0 introduced `yield` keyword
- Easy way to implement `IEnumerable` or `IEnumerable<T>`
 - No need to create custom implementation of `IEnumerator`
 - `yield return` returns next value in iteration
 - `yield break` steps out of iteration loop

C# 3.0: Lambda Expressions

- Expressive power of functional programming
- Natural progression on anonymous methods
- Compact syntax to express methods
 - Very handy when writing queries in dot notation
- Allows explicit and implicit typing

```
n => (n-1)*(n-2)
s => s.ToUpper()
(int x) => x++
(x, y) => x ^ y
(x) => { return x++; }
() => (new Random()).Next(100)
```

Func generic delegate types

- For convenience sake several generic delegates are defined: `Func<...>`
- Serve as “parametrized function” variables
- No need to define every other delegate type

```
public delegate TR Func<TR>( );  
public delegate TR Func<T0, TR>(T0 a0);  
...  
public delegate TR Func<T0, T1, T2, T3, TR>(T0  
a0, T1 a1, T2 a2, T3 a3);
```

- Return type defined last

Func delegate types example

- `Func<int, bool>` is shorthand for
`public delegate bool Func(int a0);`

```
// Initialized with anonymous method
Func<int, bool> del =
    delegate (int a0)
    {
        return a0 % 2 == 0;
    };
```

```
// Initialized with lambda expression
Func<int, bool> del2 = a0 => a0 % 2 == 0;
```

Implementation of Where operator

- Taken from Sequence.cs source code under C:\Program Files\LinQ Preview\Docs

```
public static IEnumerable<T> Where<T>(
    this IEnumerable<T> source,
    Func<T, bool> predicate) {
    // ...
    return WhereIterator<T>(source, predicate);
}
static IEnumerable<T> WhereIterator<T>(
    IEnumerable<T> source, Func<T, bool> predicate)
{
    foreach (T element in source) {
        if (predicate(element)) yield return element;
    }
}
```

C# 3.0: Object initializers

- New way to initialize an object on creation

- No need to provide or use complex constructors
- No need to set individual properties or fields

- Object initializer takes any number of accessible fields or properties

```
Blog b = new Blog
{
    Name = "Alex Thissen",
    SubTitle = "Disassembling my brain" };
```

- Name of field/property must be specified

C# 3.0: Collection initializers

- Compact initialization of collections
- Collection must implement `ICollection<T>`

```
List<string> names = new List<string>  
    { "Nicole", "Lieke", "Alex" };
```

- Can be combined with object initializers

```
List<Blog> blogs = new List<Blog> {  
    new Blog { Name = "Alex Thissen" },  
    new Blog { Name = "Mike Glaser" }  
}
```

Creating projections

- Tuples are multi-valued sets of fields
- Projections are subsets of existing tuples
- Anonymous types allow ad-hoc definition of new types
 - No need to create types for every projection
- Useful for
 - combining several data sources
 - compact subsets for reporting

C# 3.0: Anonymous types

- Projections ask for

- subsets of existing types
- combinations of data from multiple types

- Anonymous type defined by `new` keyword without a class name

- Compiler creates a new class without a programmer chosen name

- For example `<Projection>f__4`

- Type is scoped to method that declares type

Initializing anonymous types

- New types can be initialized in two ways
 1. Object initializer syntax
 - Properties takes name of initializers
 2. Projection initializer syntax
 - New properties selected by name of field or property

```
Blog b = Blog.GetByID(1337);  
Posting p = b.GetLastPosting();  
var x =  
    new { Blog = b.Name, LastPost = p.Body };  
var y = new { b.Title, b.Name, b.Created };
```

C# 3.0: Local variable type inference

- C# 3.0 compiler can infer types from initializer assignments

- `var` keyword indicates compiler inferred type

- Still strong-typed
- This is not like JavaScript `var` or VB6 Variant

```
var a = 10; // Simple types
var x = new {
    Blog = "athissen", Created = DateTime.Now
}; // Anonymous types
```

- Essential when using anonymous types

Demo: Putting it all together

- Linq to SQL revisited
- Projections with anonymous types
- Local variable type inference
- Projection initializers

Linq deferred

Expressions on Linq

Another type of data sources

- A Linq data source can actually implement one of two interfaces
 - IEnumerable<T> - or -
 - IQueryable<T>

```
public interface IQueryable<T> :  
    IEnumerable<T>, IQueryable, IEnumerable  
{  
    IQueryable<S> CreateQuery<S>(Expression exp);  
    S Execute<S>(Expression exp);  
}
```

- Create deferred query execution plan

IQueryable<T>

- Generalization of query mechanism

- Source of Linq query can be any implementor

- Implement query operators as expressions

- System.Query.Queryable provides alternative implementation of query operators

- Transitioning to queryable

- Any IEnumerable<T> can be IQueryable<T>
- ToQueryable extension method supplied on IEnumerable<T>

Lambda expressions revisited

- Lambda expressions can represent either IL code or data

- `Expression<T>` makes all the difference

```
Func<int, bool> lambdaIL =  
    n => n % 2 == 0;  
Expression<Func<int, bool>> lambdaTree =  
    n => n % 2 == 0;
```

- Compiler handles `Expression<T>` types differently

- Emits code to generate expression tree instead of usual IL for delegate

Expression trees

- Expression trees are hierarchical trees of instructions that compose an expression
- Add value of expression trees
 - Actual creating of IL is deferred until execution of query
 - Implementation of IL creation can vary
- Trees can even be remoted for parallel processing

Creating IL from expression tree

• Right before execution tree is compiled into IL

```
Expression<Func<Posting,bool>> predicate =  
    p => p.Posted < DateTime.Now.AddDays(-5);  
  
Func<Posting,bool> d = predicate.Compile();
```

• Implementation of IL generation differs very much for each Linq flavor.

- Linq to SQL generates IL that runs SQL commands
- Linq to Objects builds IL with Sequence extensions methods

In review: C# 3.0 language enhancements

- Query expressions
- Extension methods
- Lambda expressions
- Object and collection initializers
- Anonymous types
- Local variable type inference
- Expression trees

Why choose Linq?

- Unified model for querying over data

- Learn it once, apply it in several situations

- Fully type aware

- Type safety from runtime
- IntelliSense from Visual Studio “Orcas”

- Option to defer execution of queries

- Manipulate expression tree before execution

- Extensible

- Create your own implementation of IQueryable

Requirements for Linq

- September 2006 CTP for Visual Studio “ Orcas”
- - or - May 2006 Linq CTP (separate installer)
- Both includes a C# 3.0 and VB 9.0 language compiler
- Currently runs on .NET runtime 2.0
- Will probably be released as part of .NET Framework 3.5

References

- Linq home page

<http://msdn.microsoft.com/netframework/future/linq/>

- Future versions of C#

<http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx>

- Future versions of Visual Studio

<http://msdn.microsoft.com/vstudio/future/default.aspx>

- Downloads

- [Linq May 2006 CTP](#)
- [Visual Studio Orcas September 2006 CTP](#)
- [Reflector](#)

Review

- Linq == querying from programming language
- C# 3.0 was created to make Linq happen
- Several applications of Linq exist
 - Linq to Objects, XML, DataSets, SQL, Entities
 - More to come in the future
- Linq approaches functional programming

Questions ?