New C#3 Features (LINQ) – Part IV

Nico Ludwig (@ersatzteilchen)

TOC

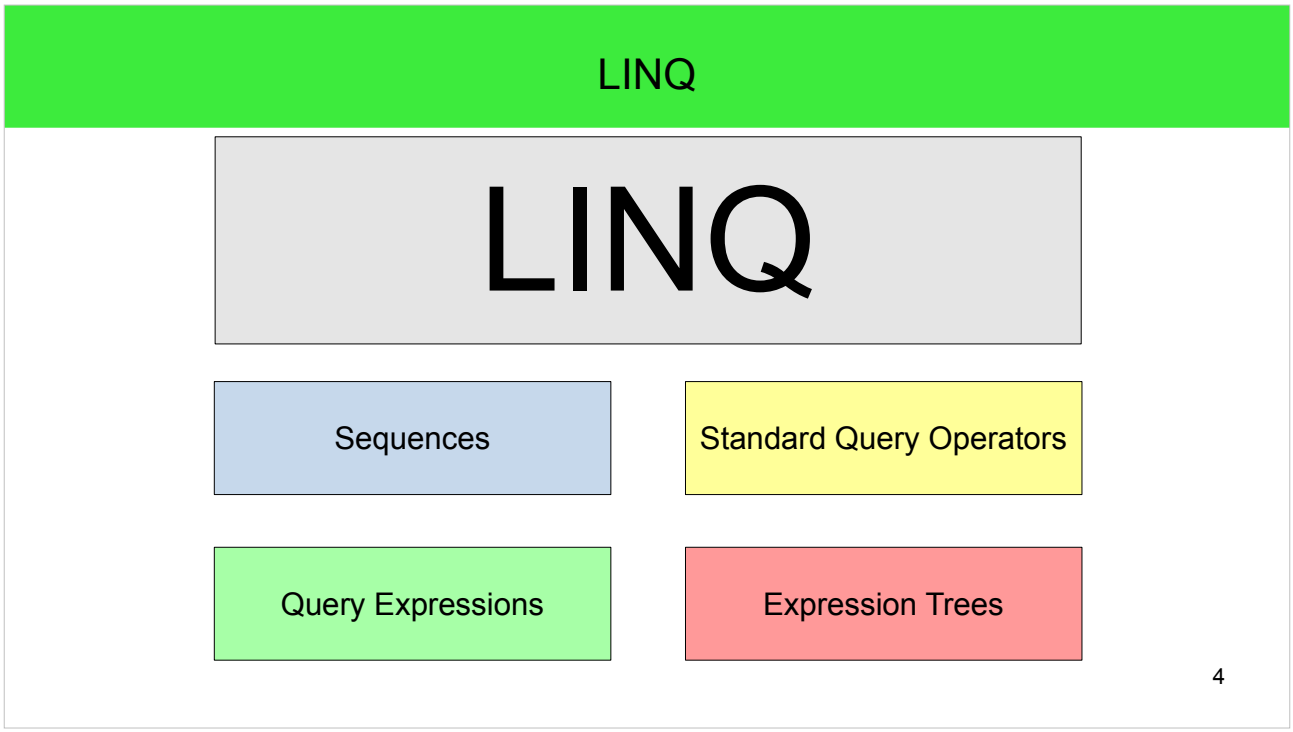
- New C#3 Features (LINQ) – Part IV
  - Introducing LINQ to Objects
  - Basic Query Expressions (Projection and Selection)
  - Anonymous Types

- Sources:
  - Jon Skeet, CSharp in Depth
  - Bill Wagner, Effective C#
  - Bill Wagner, More Effective C#
  - Jon Skeet, Edulinq, <http://msmvps.com/blogs/jon_skeet/archive/tags/Edulinq/default.aspx>
  - Fabrice Marguerie et al., LINQ in Action
  - Dariusz Parys, German MSDN Webcast "LINQ und Konsorten" of eight parts, <http://www.microsoft.com/germany/msdn/webcasts/library.aspx?id=1032369829>

LINQ's Aims

- Language INtegrated Query (LINQ):
  - Express algorithms declaratively: less code, more power.
  - Use query operators as (extension) methods, query expressions or both.
- Add a unifying syntax to the C# (VB) programming language to access data.
  - Objects and Collections (LINQ to Objects, LINQ to DataSet).
  - XML (LINQ to XML) to address impedance mismatch between C# and XML.
  - Databases (LINQ to SQL, LINQ to Entities) to address O/R impedance mismatch.
- Support deferred execution of queries.
- Support functional programming w/o side effects.
  - LINQ is only designed to query data.
  - But LINQ is able to generate new data from the queried data.

- LINQ to DataSet is similar to LINQ to SQL, but all the data is held in memory, which explains is close analogy to LINQ to Objects. .NET DataSets are disconnected in-memory representations of relational data.
- XML Impedance mismatch: Objects need to be mapped to XML.
- O/R Impedance mismatch: Object vs. Entity, identity, state, behavior, encapsulation differ completely.
- LINQ to SQL and LINQ to Entities try to address O/R impedance mismatch with generated schema and mapping models optimized for LINQ.
- LINQ to SQL (formerly known as DLINQ) is only functional with MSSQL 2005/2008 at the time of this writing.
- LINQ to Entities (was provided w/ SP1 of .NET 3.5) works with any database that implements an ADO provider model. So it is an ADO.NET based technology and allows to create higher-order entity models. It's the base for the ADO.NET Entity Framework, which may be applied with many kinds of data sources, even RESTful web services (it is queried via Entity SQL (ESQL)).
- All CRUD (Create, Read, Update, Delete) operations are possible in LINQ to SQL/Entities. It is covered by methods like *InsertOnSubmitChanges()* or *DeleteOnSubmit()*, e.g. provided in the type *System.Data.Linq.Table* or the LINQ to Entities context. These operations are type driven (pass a *Car* object to *InsertOnSubmitChanges()*, then an object with that *Car*'s identity is inserted in the data source).
- History: LINQ was inspired by the Microsoft Research projects Cω (providing, streams/sequences, query expressions, anonymous types and XML literals) and X# /Xen (providing XML extensions).



- Let's assume that syntactical enhancements like the new ways to initialize objects and collections, anonymous types, lambdas and implicit typing contribute to the column "Query Expressions".

LINQ's Query Expressions are plain C#!

- LINQ's query expressions are built on top of two concepts:
  - 1. A query language, the query expressions, represented as a set of C# keywords.
  - 2. A translation of query expressions into a set of methods, the query operators.
    - The C# compiler performs this conversion.
  - (C#'s LINQ code resembles a mighty 4GL (like, e.g., Visual FoxPro or Progress).)
- Query operators can be instance or extension methods.
  - A set of eleven query operators makes up the platform for query expressions.
  - So a programmer can write own implementations of them.
- The .NET framework's query operator implementations come in two sorts:
  - As extension methods of *System.IEnumerable<T>* (LINQ to Objects).
  - As extension methods of *System.Linq.IQueryable<T>* (LINQ to SQL/Entities).
  - (The expressiveness of .NET's query operators is improved by query expressions.)

- Query expressions are part of a specific .NET language (i.e. add specific syntax extensions to C# or VB).
- Query operators are no language extensions, but extensions of the Framework Class Library.
- In .NET 3.5 there exist 51 query operators.
- Only certain overloads of certain query operators will be translated to C# query expressions. – This is the virtual integration into the language.
- *IEnumerable<T>* and *IQueryable<T>* query operators differ considerably in how they work. *IEnumerable<T>*'s variants use delegates and deferred execution and *IQueryable<T>*'s variants use expression trees and remote execution. *IQueryabla<T>* uses the type *Expression* instead of the delegate types *Action* or *Func*. *Expression* objects can be compiled, serialized and passed to remote processes. These features are important as they allow expression trees to be used with *IQueryable<T>*, i.e. against remote databases.
- Another way to understand *IQueryable<T>*: It defines delay-compiled queries, whose data is not in memory (but, e.g., on a database), or which needs to optimized before it is executed.
- *IQueryable*:
  - 1. Lambdas are parsed to expression trees. (Code as data!)
  - 2. Query providers try to translate the expression tree to SQL (e.g. T-SQL).
  - 3. The SQL is transmitted to the database engine that executes the SQL.

How LINQ to Object maps to Query Operators

See accompanying project <LINQExtensionMethods>
Shows how LINQ to Object maps to IEnumerable<T>'s extension methods as query operators.

- These examples show how LINQ actually integrates itself into the language.

Functional and Declarative Expressibility

- LINQ to Objects is based on extension methods of *IEnumerable<T>*.
- One can get a good understanding of LINQ when comparing it to other tools:
  - Regular expressions, SQL, XPath, XQuery (including FLOWR) and XSL.
  - Functional programming languages, such as Scala or F# (new in VS2010).
- => These examples are mighty declarative languages that are worth learning.
  - LINQ is the next tool .NET programmers should get familiar with.
- Declarative languages can be hard to learn for "non-functional" programmers.
  - Spend some time to think about how problems can be solved with LINQ.
  - Try to tackle the nuts and bolts of LINQ as well as C#'s features supporting LINQ.
  - Read examples and exercise yourself.
  - => In future C# (and VB) programmers will have to know LINQ!

- There also exists the programming language Pizza (a Java extension). – It uses the concept of "fluent interfaces" to query data in a declarative manner and that is in principle identical to LINQ.

Some Details about the LINQ Translation Process

- Query expressions have to obey the LINQ query expression pattern.
  - The translation from query expressions to method calls is done iteratively.
  - First, all parts of a query expression will be syntactically mapped to method calls.
  - Then the compiler will search for matching query operator candidate methods (instance/extension/generic/overload) depending on the queried object's type.
- You can provide "your" where query expression keyword as extension method!
  - The compiler can't verify what "your" *Where()* query operator does, only the correct syntax/signature is "mechanically" checked (exactly as for .NET interface-contracts).
- Query operators can request delegates or expression trees as their parameters because lambda expressions are convertible to both.
- Ensure that your query operators match the .NET defaults in behavior.
  - Except you can provide a better solution for your own types.

- There is a complicated order and other actions involved.
- E.g. one can provide an own implementation of the query keyword where as extension method. – So the use (and need) of extension methods should get clearer.
- The definition of custom functions being matched to query expression keywords is similar to operator overloading.
- Extension or instance methods can be used, because both provide an equivalent invocation syntax.
- The C#3 standard provides a definition of the query operators that must be implemented and how they must be implemented to support the full query expressions pattern. -> If that operators are implemented as extension methods, you have just to make sure to add a using directive to your defining namespace in addition or instead of *System.Linq*.
- To be able to use the query operator patterns you can provide implementations different from extension methods of *IQueryable*<*T*> or *IEnumerable*<*T*> (e.g. reactive extensions (extension methods on *IObservable*<*T*>) or PLINQ (extension methods on *IParallelEnumerable*))
- Warning: mixing user defined and standard query operators (as extension methods) can lead to ambiguities (it's "namespace-controlled").
- In VB 9 more query expression keywords are mapped to query operators than in C#!

Query Expressions in C# – The big Picture

```
from [ type ] id in source
[ join [ type ] id in source on expression equals expression [ into id ] ]
{ from [ type ] id in source | let id = expression | where condition }
[ orderby ordering, ordering, ... ]
select expression | group expression by key
[ into id query ]
```

- To play with LINQ the REPL-tool "LINQPad" can be recommended.
  - http://www.albahari.com/linqpad.html

- Looks like SQL being written turned upside down. This is because a query expression's sub expressions are really evaluated in the order they were written (from clause then the filter clause and then the projection).
- In many (functional) programing languages (Erlang, Scala, Python and F#) you can find so-called set- or list-comprehensions that apply a similar syntax to simplify operations on sequences of data.
  - E.g. in Common Lisp there exists the loop macro for list comprehension, which has comparable abilities and clauses-based syntax.
- The from and select|group clauses are mandatory.
- There are two ways to express projection: select and group.
- The query expressions of VB 9 are more elaborated.
- Some folks count LINQ query expressions to the so called internal DSLs. But this is not 100% correct. – Only the query operators are based on language features being already present, so they are more or less internal DSLs. For the query expressions new keywords and the compiler driven transformation from query expression to calls of query operators have been introduced.
- With LINQPad you can quickly try out LINQ query expressions. It is a read-eval-print loop (REPL) tool, like the top level in Lisp.

Basic Building Blocks of Query Expressions

- The mandatory from clause specifies the sources to query data from.
  - It declares the range variable(s) and specifies the collection to query data from.
  - Range variables allow code completion and can be declared explicitly or implicitly.
  - Multiple from clauses generate cartesian products or "flat outs" of the sequences.
- The where clause defines a selection or filter.
  - The count of input items need not to be equal to the count of output items.
- The select clause defines a projection.
  - It selects the data of the range variable(s) (see: from) that will be part of the result.
  - The count of input items is equal to the count of output items.
  - (A pending select or group by projection must be used in all C# query expressions!)
  - ... but in degenerate selects that selects can be optimized away by the compiler.
  - Anonymous types can be used to reduce the need for sole projection types.

- Esp. because the from clause comes first, VS' IntelliSense is fully functional on the introduced range variables and can be used immediately in the where and select clauses.
- Multiple range variables introduced by from clauses are accessed via transparent identifiers in query expressions underneath. – The resulting *SelectMany()* call is completely streamed.
- Cartesian products are the results when the right sequence is independent of the current value of the left sequence. Flat-outs are the results when the right sequence is dependent on the current value of the left sequence.
- Multiple where clauses can be chained, in the result the predicates of the where clauses will be and-joined.
- For example where: as *IEnumerable<T>* extension method in LINQ to Objects, the predicate in the where clause would be executed as literal C# code for each row of the input sequence in the worst case. But as extension method of *IQueryable<T>* in LINQ to SQL an expression tree would be created that will be translated into highly optimized/reduced T-SQL code (incl. an optimized where clause) being executed on the database.
- Besides where the query operator *OfType()* can be used to filter items.
- You can find projections like the select clause in virtually all functional programming languages; often this function is called "map".
- There are two ways to express projection: select and group.
  - The expressions being used in select|group clauses must be non-void expressions, i.e.they need to produce values as results of the projection! – It means you can not just call a void method in a select|group clause. – What projection should be created and to which type should the result be inferred if a void-expression is used?

Anonymous Types Examples

See accompanying project <AnonymousTypes>
Shows the usage of anonymous types (with cascading and in arrays).

Anonymous Types – Part I

- Combine the creation of a new unnamed type with the creation of an instance.
  - They remove the need for intermediate types that pollute non-local code.
  - You get extra features because the compiler can generate appropriate code.
- They are immutable, sealed and internal classes (i.e. reference types) that
  - support initializer syntax that must be used to create instances.
  - (E.g. their properties are always implicitly typed w/o the var keyword.)
  - have just public readonly properties, but can not have user defined methods.
  - deliver a purposeful override of *ToString()*.
  - implement *IEquatable<T>* and inherit object automatically, nothing else.
  - have value type semantics for equality (e.g. all fields are getting equality compared).
- Declarations are considered equal, if the property names/types match in order.
  - This is the case, when their declarations reside in the same assembly (internal).

- Similar to ECMAScript's object literals.
- After assigning the last item of the anonymous type, you are allowed to leave a dangling comma (you can also leave it on arrays).
- Anonymous types are mutable in VB.
- Confusingly the properties of anonymous types any another ways to express implicitly typed variables in C# (other ways: the let clause, the var keyword, and arguments of generic methods).
- Can not refer to their own instance. The this reference does always point to the surrounding instance (in non-static contexts).
- Besides the implementation of *IEquateable<T>* an implementation of *GetHashCode()* is included.
- Possibly it makes sense to create a named type w/ an optimized implementation of *GetHashCode()* and *Equals()* so that not all properties are being compared. – In VB so called keyed properties can be defined, these readonly properties deliver the equality semantics for the anonymous type. – But in most cases it makes no sense (or is even dangerous) to create a named type, because you lose the handy stuff!
- No overloads for == and != are generated!
- The property-wise equality is also called "structural equality". This type equality is needed to create arrays of anonymously typed objects.

Anonymous Types – Part II

- The name of an anonymous type is of course unknown, so how to reference it?
  - The references of anonymous types can only be used with implicit typing.
  - 1. Use the var keyword with an object initializer in one statement.
  - 2. Or call generic methods, where the type gets inferred from arguments.
  - 3. Or use implicitly typed lambda expression parameters to carry their references.
- Anonymous types can also be cascaded.
  - This is possible, because their properties are implicitly typed (w/o the var keyword).
- Create arrays of anonymous types with implicitly typed arrays.
- Problems with the anonymous types in practical usage:
  - They are contaminative and can make code more difficult to read.
  - Its usability is very much depending from the code editor's quality (e.g. IntelliSense).

- Anonymous types should only be used for LINQ code (projections etc.), other uses are often abuse. As can be seen anonymous types do somehow "infect" the subsequent code they're used in, because you have to use implicit typing all over.
- When anonymous types are being used together with lambda functions on generic methods, the need for generic type constraints on these methods is completely removed. Constraints would hinder decoupling here.
- As mentioned frequently here we have fluent interfaces as expression of the pipes and filters pattern with extension methods on *IEnumerable*<*T*>. Normally there is one problem with this pattern: you bundle the involved components via the datatype of the piped data. But with LINQ you use anonymous types and generic methods that reduce this dependency completely. – Generic methods allow anonymous typed together with type inference to "flow" through your calls, this is just the way query operators work!
- On the other hand anonymous types are contaminative as if they're used in projections they force you to use implicit typing generally.
- At least when methods are need to be contained in a new type, you'd have to introduce a named type.
- If you're forced to use anonymous types with reflection, you're doing it completely wrong! -> Introduce a named type. But sometimes it is required to call properties of an anonymous type within another method. – For example this can be the case when you create a LINQ sequence of anonymous typed objects and bind it to a WPF *ItemSource*, then within an event handler you have to analyze the properties of, e.g., the selected item. – In this case the usage of .NET 4's DLR along with the dynamic keyword is better than explicit usage of reflection.

Thank you!