New C#3 Features (LINQ) – Part V

Nico Ludwig (@ersatzteilchen)

TOC

- New C#3 Features (LINQ) – Part V
  - Advanced Query Expressions (Ordering, Grouping, Aggregation, Joins and Subqueries)
  - LINQ Optimization, Pitfalls and Tips

- 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>

Order By, Group By and Having Clause Examples

- The last presentation stopped on anonymous types; there we'll continue discussing LINQ.

See accompanying project <MoreLINQ>
Shows the usage of projections, orderby, group by and "having"-clauses.

Ordering and Grouping with LINQ

- The orderby clause allows sorting items in a stable (LINQ to Objects) manner.
  - By default sorting is ascending, alternatively the keyword descending can be applied.
  - The order types ascending and descending can be combined on different properties.
  - The property being used with orderby must implement *IComparable*/<*T*>.
- The group by clause enables projection of grouped data.
  - (A pending group by or select clause must be used in every query expression in C#!)
  - The expression in front of by defines a projection (exactly as select does).
  - The expression after by defines the dimension or key by which to group the result.
  - The variable used after into can be used elsewhere (*IGrouping<K, T>*).
  - Each *IGrouping<K, T>* object
    - has a property *Key* (*EqualityComparer*), the group dimension (the group's common value)
    - and implements *IEnumerable<T>* to enumerate all the items of that group.
  - Additionally the method *ToLookup()* creates a flat representation of a group result.
  - The classic SQL having clause can be created with where clauses on groups.

- LINQ does not support a form of binary search. If you require this, you should resort to *IList<T>*.
- The orderby operator in LINQ to Objects is stable, i.e two equal items will be returned in the original order. Other .NET sorts algorithms are not stable. Other query providers than the one in LINQ to Objects don't guarantee stable sorting as well.
- 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?
- The order of the groups returned by group by matches the order of the found keys in the input sequence; the order of the items in each group matches the order found in the input sequence.
- The collection of grouped results with the into keyword (to use the grouped results for further post processing) is called "query continuation". Query continuations can be used with the select clause as well, but it is mostly used in group by clauses.
- The *EqualityComparer* mechanism: The type of the group's key has to implement *Equals()* and *GetHashCode()* appropriately.

Aggregate Functions, let Clauses and distinct Results

See accompanying project <MoreLINQ>
Shows the usage of aggregate functions, let clauses, transparent identifiers, distinct results and the EqualityComparer mechanism.

Other Results of LINQ Expressions and Values

- Aggregation via *Count()*, *Min()*, *Max()*, *Average()* etc.
  - These query operators can't be expressed with C# query keywords.
  - Here extension methods must be used on the query results.
  - (But in VB 9's query expressions, respective query keywords are present; e.g. for *Count()*.)
  - Types used with *Min()* and *Max()* must implement *IComparable*/*<T>*.
- Making results unique with *Distinct()*.
  - Distinct projections must be done with an extension method as well (i.e. *Distinct()*).
  - The comparison is done via the *EqualityComparer* mechanism.
- With the let clause you can define values in your query expressions.
  - The let clauses introduce new range variables and allow to use intermediate values.
  - So it's a way to define readonly "variables" as part of a query expression.
  - (Values defined in let clauses must always be implicitly typed and can't be written.)

6

- Beware of *Count()* In the programming language Lisp following fact is well known: if you need to get the count of elements within a list in your algorithm, then you've already lost the game. Also mind how the foreach loop is handy w/ sequences, as it doesn't need to know the length of the to-be-iterated Collection/sequence.
- *Distinct()*: The order of the items in the resulting sequence is undefined. When there is a couple of equal items, it is undefined which one will be put into the result sequence. VB has an extra query keyword for *Distinct()*.
- The *EqualityComparer* mechanism in *Distinct()*: The type has to implement *Equals()* and *GetHashCode()* appropriately.
- Confusingly the let clause is another way to express implicitly typed symbols in C# (other ways: the var keyword, properties of anonymous types and arguments of generic methods).
- The C# language calls the symbols defined with let clauses as "readonly range variables". – This indicates that they are being filled on each iteration of the query as the range variables in the from clauses are. => The value in the let clause will be filled on each iteration of the query.
- In fp the term "value" is used for such items being non-modifiable after initialization; the term "variable" would be quite in adequate! In most fp languages values are the common case when you define symbols. To define "real" variables that allow assignment and all other sorts of side effects being performed on them more syntactical fluff is required typically (i.e. as in F#).
- The values introduced by let clauses are accessed via transparent identifiers in query expressions underneath.

Joins, Subqueries and nested Queries

See accompanying project <MoreLINQ>
Shows the usage of joins, subqueries and nested queries.

Joining, Subqueries and nested Queries

- LINQ enables joining of related data.
  - LINQ allows to join multiple sequences on common property values.
  - Inner and outer joins (with group joins) can be expressed with query expressions.
  - The order of the expressions in from and join clauses controls right/left join direction.
  - Only equi joins are supported (based on the *EqualityComparer* mechanism).
  - For other sorts of join, anonymous types and where clauses must be used.
  - It is also possible to join data from different LINQ providers (e.g. objects and XML).

- Subqueries and nested queries:
  - Subqueries mean to use query expressions as input for other query expressions.
    - Can also be used to express query continuation (outer query continues inner query).
  - Nested queries mean to compose queries by other queries, so that the query expression resembles the produced result.
    - This concept is very important for LINQ to XML, where it is called functional construction.

- The *EqualityComparer* mechanism: The type of the join's key has to implement *Equals()* and *GetHashCode()* appropriately. If you want to join for multiple keys you have to use an anonymous type (as for group by).
- On joining with the join keyword (SQL-92 like) the order of the from-in/join-in and on/equals must match. (The SQL-92 syntax and the VB syntax are more forgiving here, however...)
- On joining with the where keyword (SQL-82 like) the order of the arguments of the == operator does not matter.
- When to use the join keyword and when to use a couple of from keywords with the where keyword?
  - In LINQ to Object queries, join is more efficient as it uses LINQ's Lookup type.
  - In LINQ to SQL queries, multiple froms (i.e. *SelectMany()*) are more efficient.
  - *SelectMany()* is more flexible as you can use it for outer- and non-equi-joins, and also the order of the being-compared expressions doesn't matter.
  - But in the end: choose the syntax that is most readable for you!
- The query operator *Join()* will eagerly read the right sequence completely and buffer it to avoid repeated iteration on that sequence. – But it still executes deferred, as this buffering of the second sequence will happen when the result sequence is started to be read. The query operators *Except()* and *Intersect()* follow the same pattern. (In opposite to that the query operators *Zip()*, *Concat()* or *SelectMany()* are completely streamed.)

Non-generic Collections and Result Materialization

See accompanying project <MoreLINQ>
Shows the usage of LINQ with non-generic collections (sequentialization), result materialization and how to deal with "dangerous" query operators.

LINQ Tips – Part I

- Sequentialize: LINQ with no-sequences/non-generic/weakly typed collections.
  - For non-generic collections use *Cast<T>()* or *OfType<T>()* to get *IEnumerable<T>*.
- Materialize, if evaluating the same LINQ will regenerate the same sequences!
  - Perform the normally deferred execution of a *IEnumerable<T>*/*IQueryable<T>* instantly.
  - Materialize with *ToArray()*, *ToList()*, *ToLookup()* or *ToDictionary()*.
- I don't know how to use LINQ to get my result! – Don't despair!
  - You can and should combine query expressions and extension methods.
  - The steps of a query expression and extension methods can be debugged as well.
  - (Not every task needs to be solved with LINQ! Solutions should always remain maintainable!)
- Query sequence results should be caught as symbol of type var, because:
  - The result of a query could be an *IEnumerable<T>* or *IQueryable<T>*.
  - The generic type *T* could be an unknown type.

- *OfType()* will filter for the specified type and items being non-null, *Cast()* won't do that. Both will only perform reference and unboxing conversions (these are only conversions, for which the cast type can be answered with true, when the operator "is" is used, i.e. no user conversions).
- Materialization (or conversion) means that the resulting collection is independent from the input sequence (as a shallow copy). Even if the input sequence is backed by an array, *List*, *Dictionary* or *Lookup*, always a new Collection will be returned (mind the "*To*"-prefix). This also means that modifications to the input sequence will not reflect in the result afterwards, the resulting collection is a snapshot! – In opposite to the conversion operators there are wrapper operators like *AsEnumerable()* and *AsQueryable()* (mind the "*As*"-prefix) that do reflect modifications of the input sequence.
- Every query expression can be expressed as C#-code using calls to query operators as extension methods. But the opposite is not true.
  - Some query operators have simply no query expression equivalent (e.g. *ToArray()*).
  - You can't use all kinds of overloads in query expressions.
  - You can't use statement lambdas in query expressions. (This is the cause why object/collection initializers have been introduced into the C# language.)

LINQ Tips – Part II

- Don't use sequence-greedy operations (where, orderby) on huge sequences!
  - Front load *Skip()* and *Take()* query operators to perform a paging.
- Front load any where filters to reduce the amount of data.
- Influence rigid LINQ queries at run time.
  - Exploit variable capturing and modified input sequences to reuse queries.
  - Exploit extension method chaining to build queries at run time.
  - Exploit expression trees to build queries at run time.
- !! In the end query expressions and extension methods are the preferred ways to express algorithms against Collections by .NET 3.5 !!

- Operations being greedy on the passed sequence need to read the complete passed sequence and can't interrupt their execution untimely.
- VB has extra query keywords for *Take()*, *TakeWhile()*, *Skip()* and *SkipWhile()*.
- In the resources you'll find extra examples that show how to exploit variable capturing and how to build dynamic queries at run time. The idea is to modify the rigid "steps of a LINQ recipe" to influence the result of a query rather than only influence the result by the input sequence.

LINQ Tips – Part III

- Consider using *ToLookup()* instead of *ToDictionary()*: *ToLookup()* allows multiple values for the same key.
  - On the other hand the result of *ToDictionary()* is mutable, *ILookup* is not.

- Don't return sequences from properties, but only from methods.
  - Providing sequences via methods underscores, that each method-call will return a new sequence.
  - Only use Collection types as property types to stress the "solidity" of the result.

- Don't materialize only to call *ForEach()*! Use foreach instead!
  - This is a waste of memory and some waste of run time performance!
  - Materialization and the *ForEach()* call both do an inner loop, with a foreach statement only one loop for all is required.
    - Materialization will execute and materialize the sequence and *ForEach()* will iterate the result, foreach provides execution and iteration in only one iteration. → The materialization step is superfluous!
  - The *ForEach()*-call might be feed with an anonymous function, which is another extra object.
  - The *ForEach()*-call-construct cannot be broken/continued. In foreach we can use break and continue for flow control.
  - => Keep the sequence a sequence as long as possible.
  - => Maybe the only reasonable use of *ForEach()* is calling an already present *List* (not just materialized) together with a method group!

LINQ to XML

- LINQ can not only be used with relational data.
  - It is possible to query and generate XML with LINQ intuitively and easy.
  - (Functional construction by using nested query expressions and syntax indenting.)
  - This idea is also present in the JVM-based programming language Scala.
- LINQ to XML Is almost able to replace XPath 2 and XQuery.
  - XPath and XQuery are used as opaque strings, but LINQ to XML is real C# code!
    - E.g. LINQ to XML can be commented and debugged, but is still as compact as a literal.
- LINQ to XML introduces new .NET types handling XML.
  - This new API is a more expressible alternative to DOM.
  - XML creation code does often obscure how the XML would look like if viewed (appropriately formatted) in a text editor. The new XML types address this issue.
  - LINQ to XML was taken to the next level in VB.NET, where it has an extra syntax.
    - VB's way is similar to the Scala approach by introducing XML literals.

13

- LINQ to XML is able replace XPath2 and XQuery.
  - Well, that was the aim; LINQ was integrated into the .NET language respectively. It is not a contaminant.
  - ... and it is compiler checked.
- There is more: As an MS research project Reactive Extensions (Rx) implements a push/subscription-based implementation of the standard query operators on *IObserver<T>* and *IObservable<T>* (available in .NET 4) to deal with asynchronous datasources (whereas *IEnumerable<T>* is pull-based).
  - Think: LINQ to Events. E.g.: perceive Button as a Collection of Click-Events. You'd have something like "from *e* in *events* where *e.Position.In*(2, 3, 20, 20) select *e*", i.e. you filter for events instead of using event handlers.
  - This is interesting if data producer and consumer work at different speed. Rx was developed by the MS Cloud group, were this is typically the case. Rx makes concurrency accessible.
  - Rx does not support all standard query operators, because some of them, i.e. the buffering ones, make no sense in a push-based world.
- In .NET 4 there is also Parallel LINQ (PLINQ) available for parallel execution of query operators (i.e. implementations of the standard query operators on IParallelEnumerable<T>).
- VB's support of LINQ to XML is also similar to ECMAScript's "ECMAScript for XML" (E4X).
- There exist serializers and other libraries (not from Microsoft), which map (there are different ways to map) LINQ to XML's *XDocument* to JSON.

Disadvantages of LINQ

- Understanding deferred execution.
  - It is an easy task to introduce bugs due to deferred execution.
  - It is a hard task to debug code applying deferred execution.
- Programmers need to know a bunch of C#3 features to employ LINQ effectively.
  - Implicit typing and complex queries can produce strange compiler errors.
  - Esp. dynamic queries are not that intuitive.
  - Just using different namespaces can influence the behavior of queries (extension methods).
- Algorithms written in LINQ (Language INtegrated Query) can not be easily migrated to other languages.
  - This is especially true for non-.NET languages.
    - Then the LINQ code must be translated to a procedural implementation.
- Warning: the big benefit of LINQ is expressibility, not performance!

- Esp. due to multiple execution of the same query with modified context/closure etc. different results can merge. – Often this was not desired and is a bug. – It is not an fp style of programming to use side effects.
- In a sense the complexity of these compiler error messages are comparable to the compiler error messages produced with C++/STL code.

Not discussed Features new in C#3 and .NET 3/3.5

- Primarily for code generation:
  - Partial methods
- Remainder of the LINQ toolset:
  - Query providers, *IQueryable*, the *LinqDataSource* component available in ASP.NET.
  - LINQ to SQL, LINQ to Entities
  - LINQ to DataSet
  - LINQ to XML
- New APIs:
  - Windows Presentation Foundation (WPF)
  - Windows Communication Foundation (WCF)
  - Windows Workflow Foundation (WF)
  - Windows CardSpace

15

- Partial methods are e.g. important for customizing code on generated data model classes for LINQ to SQL. They are somewhat similar to the concept of the GoF pattern "Template Method".
- *LinqDataSource* (LINQ to SQL) and *EntityDataSource* (LINQ to Entities) allow to perform ordering, paging, editing etc. of web contents. E.g. as two-way bound *DataSource* on an ASP.NET GridView. You can always use LINQ sequences (e.g. from LINQ to Objects) as one-way bound *DataSource*s in ASP.NET.
- .NET *DataSet* are disconnected in-memory representations of relational data accessible via LINQ to Objects.
- CardSpace is Microsoft's implementation of InfoCard, which is dead meanwhile. InfoCard is a technology for user centric security, but meanwhile there are other technologies like OpenID.
  - Windows CardSpace is a system to manage different identities against other environments, e.g. web sites. CardSpace is like your wallet with different cards representing different identities.

Thank you!