New C#3 Features (LINQ) – Part III

Nico Ludwig (@ersatzteilchen)

TOC

- New C#3 Features (LINQ) – Part III
  - Algorithms in C#3
  - Extension Methods on IEnumerable<T>
  - Deferred Execution

- Sources:
  - Jon Skeet, CSharp in Depth
  - Bill Wagner, Effective C#
  - Bill Wagner, More Effective C#

Algorithms with C++ STL

See accompanying project <STLAlgorithms>
Shows how an algorithm operating on data can be expressed in C++11 with STL.

Problems with STL in Practice

- Yes, the STL serves its purpose, but there are many problems in practice.
  - The independent begin and end iterators must be used/interpreted as sequence.
  - Using multiple STL functions to get a consecutive result, means that we have to pass iterators all over to all that functions to perform side effects.
  - Some STL functions return new iterators, which must be used instead of end to build new sequences, others don't.
  - => This leads to a non-uniformed coding style with STL.

- To be frank most problems arose, because STL has to work with legacy data as well.
  - The idea of iterators was to separate data from algorithms.
  - Esp. they work with bare arrays, which is very cool indeed!

- In fact the STL is very error prone, and its syntax is not appealing.
  - Even experienced programmers explore the STL often with the debugger...
  - Implementing own, STL compatible functions can be very hard.

Algorithms with IEnumerable<T> Sequences

See accompanying project <ExtensionMethodsAlgorithm>
Shows how an algorithm operating on data can be expressed in C# with IEnumerable<T> sequences, extension methods and chaining.

- The chaining notation, which is a feature that came with the introduction of extension methods, allows to write the method calls in the order in which they are being applied.
- Chaining means that the result of a method is used as input for another method.
- The jQuery library also applies chaining notation to operate on so-called "matched sets" (instead of sequences) of markup elements.
- In the Smalltalk programming language all methods return a reference to the "this" reference ("*self*" in Smalltalk) by default to support fluent interfaces. Additionally Smalltalk provides a special chaining syntax operator. If a method does not return self, the special method "*yourself*" can be used in the chain to return the self reference of the object used as "origin" of the call chain.
- 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 the *IEnumerable<T>* algorithm model. The ideas tested in Pizza have been put into effect in Scala.

IEnumerable<T> Sequences versus the STL Design

- What do they have in common?
  - The results of STL iteration and sequences are ordered.
  - Both try to separate data from algorithms.
  - The declarative style describes the result, not the steps how to create the result. → There are no loops.
    - (Java 8 introduced the terms internal (*Count()*) vs. external iteration (explicit for loops).)
  - Aggregate functions that produce scalar results are available (e.g. *Count()*, *std::count()*).
- With generic Collections in .NET 2, algorithms have only been improved barely.
  - This lack has been addressed with *IEnumerable<T>*'s extension methods in .NET 3.5.
- *IEnumerable<T>*'s extension methods are easier to read/understand and write.
  - The separation of data and algorithm is applied via *IEnumerable<T>* sequences.
  - Extension methods have no side effects on input sequences, but produce new sequences.
  - In effect *IEnumerable<T>* sequences are returned as result, not any arcane iterators.
  - Consecutive results are combined via the simple and uniformed chaining notation.
  - It can be compared to the pipe functionality of different shells (e.g. bash or PSL).

- Virtually only objects implementing *IEnumerable<T>* are called sequences in LINQ parlance, not those objects implementing the weakly typed interface *IEnumerable*.
- The chaining notation, which is a feature that came with the introduction of extension methods, allows to write the method calls in the order in which they are being applied.
  - To be fair in C++ there is an API that uses a chaining notation: the STL iostream API (what a matching name...).
- The idea to have a set of independent functions that can be combined to operate on data as a sequence is often called the "pipes and filters" pattern.

From old Style to new Style Algorithms

See accompanying project <DeferredExecution>
Shows the evolution of an old style algorithm to an algorithm applying deferred execution.

.NET Algorithm Model on IEnumerable<T> - Part I

- All .NET Collections have one thing in common: the interface *IEnumerable*/*<T>*.
  - (Exactly as in C++' STL, containers and arrays provide the common idea of sequences as a range between two pointers, called iterators.)
- Originally *IEnumerable<T>* itself was designed for iteration only,
  - ...but *IEnumerable<T>* was augmented with additional extension methods.
  - As interface-extension methods on *IEnumerable<T>*, all implementors are affected!
  - These extension methods are available for all .NET Collections and arrays instantly!
  - Our types implementing *IEnumerable<T>* will support these extension methods for free.
  - (*IEnumerable<T>*'s extension methods are implemented in the static class *Enumerable*.)
- The idea of iterator blocks is based on *IEnumerable<T>* as well.
  - Do you remember the yield return/yield break statements?
  - The key for so called "deferred execution": produce data only when it gets requested.

- Strongly typisized collections support *IEnumerable<T>* (i.e. sequences), others *IEnumerable*.
- Iterator blocks enable deferred execution: no line of code in the body of the iterator block runs until the first call to *MoveNext()* is performed.
- Deferred execution means that the input sequence isn't read until it has to be. And the resulting sequence of deferred execution can be an input sequence for the next method.

.NET Algorithm Model on IEnumerable<T> - Part II

- In everyday programming, loop constructs are often used with Collections.
  - Here Collections make up the input, and they're being manipulated somehow.

- This loop-iteration strategy is effective but somewhat inefficient.
  - Often there are several transformations to do on each item.
  - Often (large) intermediate result-Collections must be created.
  - Finally each iteration must be completed before the next iteration can start.

- C#'s iterator blocks can optimize this by operating only on the current item.
  - Use yield return in concert with the type *IEnumerable<T>*.
  - Its calls can be chained, because they extend and return *IEnumerable<T>*.

- In the end we declare a set of steps to be executed as a recipe.
  - These steps get executed in a deferred manner, when the result iteration begins.

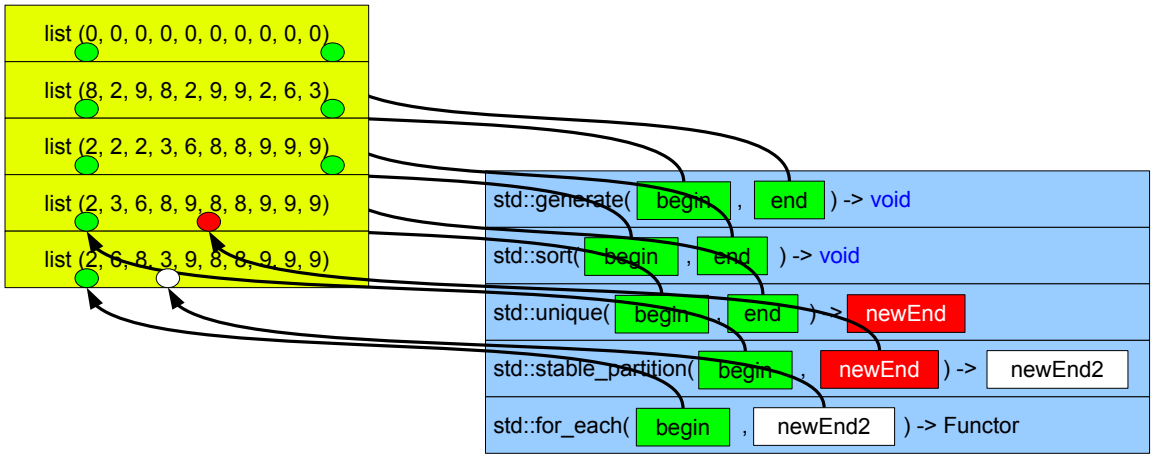
.NET Algorithm Model on IEnumerable<T> - Part III

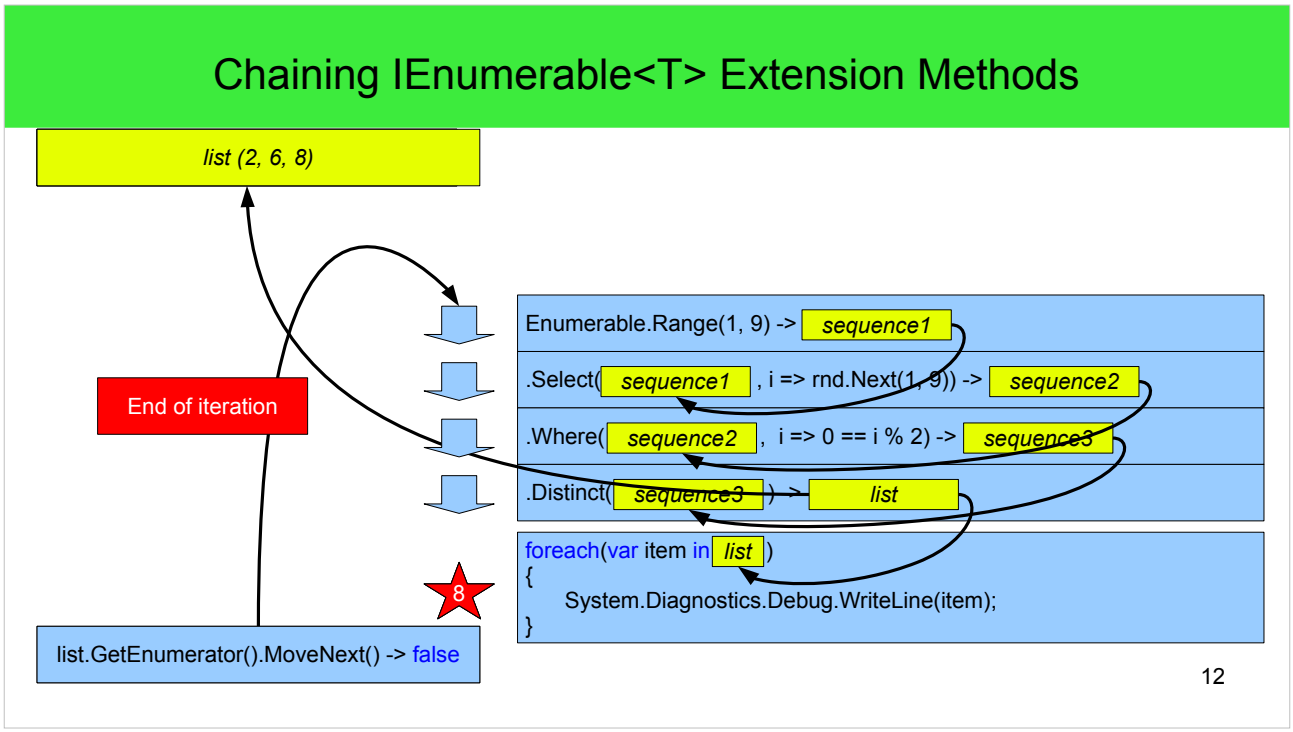
- Using iterator blocks and *IEnumerable<T>* that way needs a shift of thinking.
  - In iterator blocks *IEnumerable<T>*'s are inputs (held in parameters) and outputs (return).
  - The iterator blocks don't modify the input sequence, but produce a new one.
  - These iterator blocks act as building blocks that promote reuse.
  - Side effects are discouraged in iterator blocks (exception safety, parallelization).
  - Multiple operations are applicable in the chain while iterating the result only once.

- Benefits -> We have to express what we want to do, not how we want to do it!
  - Better reusability, because self-contained iterator blocks are used.
  - Self documentation by design: a method's name expresses its sole responsibility.
  - Better composability, esp. when implemented as extension method (chaining).
  - Better performance, because execution is deferred on just the current item.
  - Better chances to let libraries evolve, because just iterator blocks must be added.
  - Better suited to be parallelized, because there are no side effects.

- The chaining notation, which is a feature that came with the introduction of extension methods, allows to write the method calls in the order in which they are being applied.
- The *IEnumerable<T>*s as inputs will be iterated only once in such a function, because some iterators can be only iterated once! – So we should do it in our own functions as well. In some situations two sequences will act as input, then some functions must eagerly read the second sequence completely and buffer it to avoid repeated iteration on that sequence (e.g. *Join()*, *Except()* and *Intersect()*, but not *Zip()*, *Concat()* or *SelectMany()*). – But such functions do still execute in a deferred manner, as this buffering of the second sequence will happen when the result sequence is started to be read.
- Another example demonstrates that the way how this algorithm model works with data in a declarative way is quite correct: database experts dealing with huge amount of data suggest not to use cursors or iterators, rather use "set-oriented operations"

Calling STL Functions





- (The arrows show the order in which the methods have been called and in which order the sequences will be iterated.)
- *Range()* will produce nine items under all circumstances, but the chained methods may reduce that count (*Where()*, *Distinct()*).
- The chaining notation, which is a feature that came with the introduction of extension methods, allows to write the method calls in the order in which they are being applied. Extension methods simplify to call methods on the returned objects the result is what we call "fluent interface". JQuery and LINQ to XML do also exploit fluent interfaces. Extension methods to built fluent interfaces can be a first step to create so-called "Domain Specific Languages (DSLs)".
- This code does explicitly describe what it does, it is really declarative code. Because the calls are chained, there is no chance that any independent information (as on C++' iterators) can be lost. The chaining syntax feels uniform, it reflects the stream of data.

Deferred or immediate Execution and lazy or eager Evaluation and the Run Time Type

- The result of most *IEnumerable<T>* extension methods is deferred.
  - Before the result is iterated, the method (chain) won't be executed (recipe of steps).
  - If the result is multiply iterated, the code of the "chain" is always executed anew!
  - I.e. their input is evaluated lazily and their result is streamed to the output.
    - This also means that a sequence needs not to be backed by a "solid" data source or Collection!
  - Aggregate functions (*Min()*, *Count()*) can't be deferred, they have an immediate result.
  - In general all methods not returning an *IEnumerable<T>* cause immediate execution.
- But some of that deferred extension methods must be evaluated eagerly.
  - These eager methods have to read the entire sequence (*Reverse()*, *OrderBy()* etc.)
  - I.e. their results are calculated eagerly and buffered internally.
  - *Distinct()* is in between as it streams and buffers data as we go (i.e. as we iterate).
- Extension methods can choose a different implementation based on the run time type.
  - E.g. *Count()* is optimized for sequences implementing *ICollection/<T>*.

13

- In other words all query operators returning *IEnumerable<T>* or *IOrderedEnumerable<T>* apply deferred execution. So the operator won't read from any input sequences until someone starts reading from the result sequence. Some operators will first access the input sequences when *GetEnumerator()* is called, others when *MoveNext()* is called on the resulting iterator. Callers should not depend on these slight variations.
- *Distinct()* is not stable: the order of items in the resulting sequence is undefined, as if there are multiple equal items, it is undefined which one will be put into the result.
- Also *ElementAt()* and *ElementAtOrDefault()* are optimized if the passed sequence implements *IList<T>*. Along with *Count()* (but not *LongCount()*!) these seem to be the only run time type optimizations present in LINQ to objects (state in October 2011).

Extended Generic Methods' Type Inference

See accompanying project <Decoupling>
Excursus: Presents the extended type inference abilities of generic methods when used with anonymous methods.

Generic Extension Methods and Anonymous Functions

- The .NET algorithm model is based on a combination of
  - *IEnumerable*<*T*> extension methods (the so called standard query operators),
  - iterator blocks (featuring yield return/yield break)
  - and lambdas used as *Delegate* instances.

- Lambdas decouple functions in functional programming (fp).
  - As .NET interfaces decouple usage from implementation in oop, *Delegate* types decouple usage and implementation in fp.
  - *Delegate*s allow looser coupling, lambdas are sufficient as *Delegate* instances (no stringent implementation of .NET interfaces is required).
    - In effect "*Delegate* contracts" are easier to fulfill than "interface contracts".
    - A Delegate symbol doesn't care about the type that provides the method it references, just the signature must be compatible.

- C#3 provides enhanced *Delegate* type inference for generic methods.
  - New in C#3: Generic types can be inferred from anonymous functions.
  - This feature of type inference is crucial for LINQ.

- Where anonymous classes were needed in Java to implement interfaces ad hoc, lambdas jump in in C#.

Improve Reusability by Decoupling

See accompanying project <Decoupling>
Shows how to gradually decouple functionality to improve reusability of methods.

Decoupling Iteration from other Activities

- We can transform our input and output sequences to *IEnumerable*<*T*>.
  - I.e. we'll get rid of Collection types (incl. arrays) in our types' interfaces.

- The iteration activity must be separated from the "performer" activities.
  - An activity is a piece of code that is injected into the iteration, e.g. as lambda expression.
  - The kind of iteration could be controlled, e.g. filtered (predicate activity).
  - The items of the input sequence could be converted somehow.
  - On each item it could be required that a side effect has to be performed.

- That separation is done by delegating non-iteration activities to other methods.
  - Methods of *Delegate* type *Predicate*<*T*>/*Func*<*T*> can express conversion or filtering.
  - Methods of *Delegate* type *Action*/<*T*> can express side effects.

Code Injection and Exceptions

- Injected code, deferred execution and exceptions.
  - The execution of injected code is also deferred to the point of iteration!
  - E.g. exceptions will happen when the result is iterated, not when it is generated.

See accompanying project <Decoupling>

Shows how injected code can disturb deferred execution, when it throws exceptions.

- The problem is that the code, in which the run time error can occur, may be executed at a different position in the code (i.e. in a deferred manner).
  - And this is a downside of decoupling: establishing a proper error handling can be hard as we don't know about the side effects of injected code.

How to create reusable Functions?

- Once again the ideas of functional programming help here:
  - Program methods without side effects, create independent functions.
  - Communicate by returned values only, never by out or ref parameters.
  - Apply *Delegate* parameters to exploit decoupling by dependency injection.
  - Use or create extension methods (on *IEnumerable*<*T*>) to enable chaining.
  - (Iterate passed sequences only once, possibly we have no second chance!)
  - For data operations: think in pipelines and single expression style.
  - We have to name our methods according to their real (at best sole) responsibility!

- Benefits of independent functions:
  - Provide best reusability.
  - They can be tested separately (referential transparency in functional programming).
  - They can be better refactored and documented.
  - They can be parallelized without problems.

19

- Mind to document if a function is lazy or eager on evaluating the passed sequence (see the discussion about *Reverse()* and *OrderBy()* some slides before).
- If we introduce generic methods, we should think twice to apply constraints. When anonymous types (introduced in a future lecture) 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.
- Functional programming works like writing a gigantic formula or editing a spreadsheet. – Maybe spreadsheet programs (like Excel) are the most used fp languages, they are even used by non-programers.
  - The spreadsheet metaphor makes clear that fp languages are difficult to debug and that it is difficult to measure code coverage.
- Referential transparency: In imperative programming, a function's output depends on its input arguments and the current state of the program. In functional programming, functions only depend on their input arguments. In other words, when we call a function more than once with the same input arguments, we always get the same output value. (I.e. we evaluate functions for their result values instead of their side effects.) Such functions can also be called pure functions.
- The easier to archive parallelization is possible due to the fact that functional algorithms share no common mutable state.

Leave Lambdas as inline Literals

- When lambdas are used, we'll end up with identical lambda code snippets.
  - In the Decoupling example we've encountered the lambda *i => 0 == (i % 2)*.
- Named methods: pro reuse <-> Anonymous functions: pro decoupling
- Don't refactor identical lambda code into a single method for reuse!
  - In fact we disable reuse of the code that used the lambda!
  - Often we want to exploit type inference gained with lambdas.
  - The compiler can only optimize inline lambda code (esp. for LINQ to SQL).
  - Expression trees are built on inline lambda code as well.
- We have to shift our notion of building blocks to methods operating on *IEnumerable<T>*!
  - Possibly as extension methods of *IEnumerable<T>* to fit the standard methods.
  - This will maximize reusability with .NET 3.5 or greater.

- LINQ to SQL will translate query expressions to expression trees. E.g. the where expressions (mapping to the *Where()* query operator) get translated to expressions, whereby lambdas can be dissolved into subtrees, but method calls must be retained as MethodCall-expressions. Then the expression tree is translated to T-SQL. During this translation *MethodCall*-expressions can not be translated, but the lambdas' subtrees can (e.g. translate C# *s.Contains("s")* to T-SQL *s LIKE "%s %"*).
- The phase of development with lambdas along with evolving anonymous types makes inline lambdas a crucial syntactical cornerstone.

Important Algorithm Methods

- Tips:
  - Write a using directive to *System.Linq* to get access to the standard query operators!
  - Chain the calls to get the desired result sequence described as "recipe".
    - The function of Collections is primarily storing and accessing data.
    - The function of sequences is primarily describing computations on data.
- *IEnumerable<T>* query operator extension methods (selection (of 51 in .NET 4)):
  - !! They never modify the input sequence, but rather produce a new sequence!
  - *Select()*, *Where()*, *Distinct()*, *OrderBy()*, *Reverse()*, *Concat()*, *GroupBy()*, *Cast()*
  - *Min()*/*Max()*, *Average()*, *Count()*, *Sum()*, *All()*, *Any()*, *SequenceEqual()*
  - *Union()*, *Intersect()*, *Except()*, *ToArray()*, *ToList()*, *ToDictionary()*, *ToLookup()*
- Sequence creation methods of static class *Enumerable* (from *System.Linq*):
  - *Range()* – Creates a sequence of ints, very handy in practice!
  - *Repeat<T>()* – Creates a sequence of n copies of an instance of generic type *T*.
  - *Empty<T>()* – Creates an empty sequence of type generic type *T*. ("Null-object" design pattern.)

21

- Because the standard query operators are implemented as extension methods, a using directive to *System.Linq* is sufficient to make them available. – They are implemented in System.Core.dll.
- No extension methods of *IEnumerable<T>*: *Empty()*, *Range()*, and *Repeat()* (as static methods on *Enumerable*), *ThenBy()* and *ThenByDescending()* (as extension methods on *IOrderedEnumerable<T>*), *OfType()* and *Cast()* (as extension methods on *IEnumerable*).
- The standard query operators throw *ArgumentNullException* (instead of a *NullReferenceException*), if the passed *IEnumerable<T>* sequence is null. This makes sense! - Because even if a query operator could be called like a member of *IEnumerable<T>*, whereby the "argument character" of the passed sequence is invisible, we could also call the query operator with the sequence as visible argument. As we see the *ArgumentNullException* is exactly correct. - We should also follow this pattern in our extension methods for the "this" parameter.
- In .NET 4 there also exists Parallel LINQ (PLINQ) available for parallel execution of query operators (i.e. implementations of the standard query operators on *IParallelEnumerable<T>*).

Thank you!