New C#3 Features (LINQ) – Part II

Nico Ludwig (@ersatzteilchen)

TOC

- New C#3 Features (LINQ) – Part II
  - Implicit Typing
  - Lambda Expressions
  - Extension Methods

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

Local Type Inference Examples

See accompanying project <TypeInferenceExamples>
Presents local type inference.

Implicit Typing of local Variables

```
// Initialization of a local
// variable in C#1/C#2:
string someData = "a string";
```



```
// New in C#3: declare someData as
// implicitly typed local variable:
var someData = "a string";
```

- Explicit typing of local variables is no longer required!
  - Explicit type names can be replaced by the contextual keyword var on initializations.
- But implicitly typed variables are still statically and strongly typed!
  - They are statically and immutably typed after initialization.
  - Their static type is inferred by the compiler, it analyzes the initializer expression.
  - Only the interface of the static type can be used on them.
  - So they do not function like the type "Variant" in COM or VB 6 or var in JavaScript!
  - (Implicitly typed constants are not available.)

- The keyword var is similar to C++11's "additional meaning" of the auto keyword.

Details about implicit Typing

- The initializer expression needs not to be a constant or literal.
  - In C#3 there exist situations, in which the static type is unknown or anonymous.
  - These anonymous types are an important feature of LINQ in C#3!

- The compiler is not able to infer all types.
  - It must be an initialization expression (inferred is the type of the expression right from =).
  - It must be only one variable being declared.
  - The compiler cannot infer the type of anonymous functions or method groups (as they are typeless expressions).
  - The initializer must not be a null literal w/o cast (also a typeless expression).

- Type inference to constant symbols (a syntax like const instead of var i.e. "const inference"), is not supported.
  - (It wouldn't be very useful, because C# only allows compile time constants of primitive types.)

- Problems with the var keyword in practical usage:
  - It can make code more difficult to read; declaration and usage should be near.
  - Its usability is much depending on the code editor's quality (e.g. IntelliSense).

Pros and Cons of implicit Typing

- Pros
  - The compiler can often infer the most efficient type (esp. for LINQ results).
  - Implicitly typed variables must be initialized, which may reduce coding errors.
  - It can improve readability, when the inferred type is obvious.
  - Less "using-namespace-directives" are needed.
  - Code can be easier modified.
- Cons
  - Can be misunderstood by VB/COM/JavaScript developers. (No duck typing here!)
  - If the right hand side expression is complex, developers can't infer the type ;).
  - There are situations, when you are simply forced to use them.
  - Sometimes inference is the best solution, but it is not obvious why this the case.
- When in doubt mind that code is more often read than written!

- Another con: The var keyword hurts the interface segregation principle (after SOLID). – It enforces a style, where on the left hand side of the assignment a non-interface type, but a very concrete type is inferred.
- A further con: confusion. Confusingly the var keyword is another way to express implicitly typed variables in C# (other ways: the let clause, the properties of anonymous types and arguments of generic methods).

Lambda Expressions first Example

See accompanying project <LambdaExpressionExamples>
Shows how lambda expressions can express Delegate instances.

Lambda Expressions as anonymous Functions

```
// Classical usage of an anonymous
// method to instantiate a Delegate:
Predicate<int> isEven = delegate(int i)
{
    return 0 == (i % 2);
};
```



```
// New in C#3: usage of a lambda
// expression to instantiate a Delegate:
Predicate<int> isEven = i => 0 == (i % 2);
```

- Lambda expressions (lambdas) are an evolution of anonymous methods.
  - Anonymous methods and lambdas can be generalized to anonymous functions.
  - (Both allow to pass code as argument to another method.)
  - You can use expression lambdas and statement lambdas.
  - (Anonymous iterator blocks are not supported.)
  - You can generate expression trees from lambdas.
  - As a block of code lambdas can be debugged!
  - During run time the "names" of these anonymous functions are obfuscated to unspeakable compiler-generated symbols.

8

- The operator => is called "lambda operator" or "big arrow" or "rocket" (taken from Ruby's "hashrocket" operator) and can be read as "goes to".
- In ECMAScript's/JavaScript's function literals (var f = function(x) { return x * x; }) are equivalent to C#'s lambdas. – So in other words, JavaScript has lambdas for ages.
- Anonymous functions can not be iterator blocks. It was counted to be too much effort for the C# compiler team.
  - Only top level methods can be iterator blocks.
  - Anonymous iterator blocks are supported in VB11!
- The unspeakable symbols are then visible in stack-traces and in reverse-engineered code.

Lambdas in everyday Usage

See accompanying project <LambdaExpressionExamples>
Shows some everyday usage examples of lambda expressions.

Lambdas as Delegate Instances for everyday usage

- Whenever anonymous methods have been used, lambdas can be used now.
  - The compiler converts lambdas to Delegate types by type inference respectively.
  - So the utility methods on Collections (e.g. *FindAll()*) can deal with lambdas as well.
  - Functions can have other functions as parameter or result => higher-order functions.
  - The types *Predicate<T>*/*Action*/*<T>* and the new type *Func<T, R>* can be used.
- Lambdas can also be used for event handler code.
- If lambdas refer local or other variables, these variables are getting captured.
  - (I.e. lambdas lexically bind the enclosing scope's locals and the this reference.)
  - Captured variables can be modified within the capturing lambda!
  - The lifetime of captured locals is extended! Captured references can't be garbage collected (gc'd), as long as the Delegate instance (backing the lambda) wasn't gc'd!
- Partial application and currying are not directly supported but can be simulated.

10

- From within anonymous methods you can refer the enclosing scope's locals and also the this reference in instance methods of reference types. So in anonymous methods locals and the this reference are bound lexically. (In opposite to dynamic binding (binding variables where a function is called and not where it is declared) as, e.g., in JavaScript.)
- The details of garbage collection of captured variables have been discussed in the C#2 review.
- Partial application and currying:
  - Partial application: Bind a function to variable or constant arguments, so that a new function will be returned that accepts less arguments than the original function. – It can be simulated by creating a new Delegate instance from a call of the bound function with a constant argument.
  - Currying (named after the US American mathematician Haskell Brooks Curry)/Schönfinkeln (named after the Russian mathematician Moses Ilyich Schönfinkel): Slash a function having multiple parameters into a function having only one parameter, but returning a chain of functions accepting the remaining parameters of the original function. – It can be simulated by cascading anonymous functions collecting the parameters and finally calling the original function. – In short a function having some parameters is converted into a function, which accepts less parameters by binding arguments to parameters.

Expression Tree Example

See accompanying project <LambdaExpressionExamples>
Shows the basic usage of expression trees.

Lambda Expressions for Expression Trees

- Expression trees allow to treat code as analyzable and synthesizable data.
  - It's not as hidden as IL code.
  - This is a way of meta programming.
- .NET 3.5 has a built-in support to compile lambdas into expression trees.
  - Mind to use the namespace *System.Linq.Expressions*!
- This is the cornerstone of LINQ to SQL to handle code as a tree of objects.
  - 1. Lambdas are parsed to expression trees.
  - 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.
- Expression trees can also be used to solve other problems.
  - E.g. you can code own query providers to exploit expression trees.

- Expression trees can be created from lambda expressions, but not from anonymous methods.
- Notice that we have used 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.

Extension Methods Example

See accompanying project <ExtensionMethodsExamples>
Shows the evolution from static utility methods to extension methods.

Extension Methods

- Sometimes you'd like to extend the set of methods a type provides.
  - But you can't do so because the type can't be inherited or is too abstract (interface).
  - Or changing a type (e.g. an interface) would break the code of present implementors.
  - Or inheritance is inappropriate: neither behavioral changes nor new fields are needed.
  - And you just have to use that type's public interface to create the extension.

- When these kinds of desires arise you'll often end up with static utility methods.

- The problem with static methods is that they don't belong to the extended type.
  - Neither C# nor Visual Studio will help you here.
  - You'll have to call and recall static methods like global functions with the "dot notation".

- Promote static utility methods to extension methods to use them like members.
  - This will solve the discussed problems.
  - With extension methods the type looks like it was extended, but it was not modified.

Pervasive Extension Methods – Example

See accompanying project <ExtensionMethodsExamples>
Shows the pervasiveness of extension methods.

The Implementation Site of Extension Methods

- Extension methods must be defined in top level static classes.
  - The leftmost parameter must be of the being-extended type, having the this keyword as prefix.
  - The defining class is sometimes called "sponsor class".
- Any .NET type can be extended.
  - E.g. sealed, "concrete" and abstract types (including interfaces).
  - Unbound and constructed generic types.
  - If an extension method adds a new overload the respective method group will be extended.
- The extension methods itself:
  - The public members of the extended type can be accessed via the "this parameter".
  - Other parameters than the "this parameter" can be used also.
  - Multiple overloads can be defined as well.
  - A result can be returned to the caller or not.
  - There are no extension properties (indexers), constructors, events or operators.

16

- Top level static classes means "not in nested static classes".
  - You can extend multiple different types in the same static class (as it is done here), but this should be considered a bad practice. You should create one static class per type you are about to extend.
- In C# you can only extend types, not specific objects (but in Ruby both is possible).
- A method group is the set of all methods found by a member lookup of a certain method name in a certain context. E.g. all overloads of a method being effectively present due to inheritance from multiple types or extension methods are part of that method's method group. Ctors do not build a method group!
- Extension methods can only access public members of the extended type; i.e. they are no "friends" as known from C++!
- If the "this-parameter" is null, then an *ArgumentNullException* should be thrown, because the "this-parameter" could also be passed explicitly (prefix notation).

The Call Site of Extension Methods

- The namespace of the static class defining the extension methods must be used.
  - That static class could be in the same namespace as the calling entity.
  - If not, a using directive to the namespace of that class is needed!
- Calling extension methods.
  - They can be used with the period/pointer operator (i.e. "infix") on all instances directly.
  - (Infix: an instance of the extended type won't be passed as first argument.)
  - (The name of the static class defining the extension methods is never used then.)
- Extension methods are pervasive on related types.
  - This means that they add candidates for method name resolution.
  - 1. Extension methods of interfaces are available for the implementing types as well.
  - 2. Extension methods of base types will extend sub types as well.

- The name of the class in which the extension method is defined is irrelevant!
- When there exist extension methods of equal signature in different namespaces, these extension methods may clash if their namespaces are used in the same code file.
- The infix notation of extension methods allows to call them in a chained manner, then they are written in the order they are being executed.
- If the "this-parameter" is null, then an *ArgumentNullException* should be thrown, because the "this- parameter" could also be passed explicitly (prefix notation).

Name Resolution and Binding of Extension Methods

- Name resolution of extension methods with the same logical signature.
  - (The namespace of the class defining the extension methods must be visible!)
  - They are compile time (i.e. statically) bound.
  - 1. Extension methods on interfaces overrule implemented instance methods.
  - 2. Extension methods on classes overrule, when they have a better matching signature.
  - 3. Extension methods on classes can't overrule instance methods of identical signature.
  - 4. Extension methods on classes can't overrule instance methods having an implicit conversion.
  - 5. Extension methods on classes can't overrule generic instance methods (best match).

- Binding to generic types.
  - Extension methods can be defined for an unbound generic type. (*IList*<*T*>)
  - Extension methods can be defined for a constructed generic type. (*IList*<*int*>)
  - Extension methods of *IEnumerable*<*T*>s are an easy way to extend LINQ.

Extension Methods as Type Augmentation

- This slide describes how extension methods could be applied to augment types.
- 1. Define the minimal interface as .NET interface.
  - Or leave the present interfaces and classes in your API as they are.
- 2. Create augmentation methods as extension methods on that interface.
- 3. Now programmers can decide to opt in the extension methods.
  - They just need to add a using directive to make the static class visible.
  - Then the extension methods are visible as well.
- (4. Eventually the extension methods could be promoted to instance methods.)

- Notice that the visibility of extension methods is controlled by using respective namespaces.
- Indeed extension methods allow extending a type's interface w/o breaking the code of present consumers. Java 8 introduced so called defender or default methods, which can be fully implemented (i.e. with a function body) in an interface definition; default methods do not break an interface's contract, but can be overridden in implementing types. Default methods make use of the template method design pattern.

Final Words on Extension Methods

- Extension methods aren't a new concept.
  - Similar to Obj-C's categories and slightly similar to mixins in Ruby, CLOS, D etc.
- Underneath the compile time custom attribute "*ExtensionAttribute*" is applied.
  - E.g. in VB *System.Runtime.CompilerServices.ExtensionAttribute* must be used.
  - From .NET 2 code, extension methods can be called with the old prefix period notation.
- Problems with extension methods in practical usage:
  - Extension methods can not be found by reflection on the extended type.
  - Its pervasiveness on name resolution can lead to unexpected results.
  - A type seems to have a varying interface, this is confusing esp. for beginners.
  - Programmers can't tell instance methods from extension methods by the bare code.
  - Its usability is much depending on the code editor's quality (e.g. IntelliSense).
  - => Can be critical in usage, because code is more often read than written!

- The idea of a mixin is to define a part of a new type by including a bundle of predefined functions. And this bundle is called mixin.
- Ruby has also so-called open classes, but this can not be simulated with C#'s extension methods.

Thank you!