Peter Himschoot en Wim Uytterspot zijn beiden verbonden aan U2U

Uitbreidingen C# 3.0 onder de loep genomen

LANGUAGE INTEGRATED QUERY

Tijdens de 2005-editie van de Microsoft Professional Developer Conference in Los Angeles lichtte Anders Hejlsberg, de chief designer van C#, de nieuwste versie toe van de C#-programmeertaal, versie 3.0. De belangrijkste uitbreidingen die we noteren zijn impliciete declaratie van variabelen, extensiemethodes, lambda-expressies, anonieme types en LINQ (.NET Language Integrated Query). In dit artikel bespreken we deze nieuwe uitbreidingen op de C#-taal in zoverre die nodig zijn voor LINQ.

Een query-instructie die in een array zoekt naar elementen die beantwoorden aan een bepaalde voorwaarde schrijven we in C# 2.0 doorgaans met behulp van een foreach-statement:

```
string[] names = { "Peter", "Jan", "Wim", "Patrick", "Ann" };
foreach (string n in names)
{
  if (n.Length == 3)
    Console.WriteLine();
}
```

Codesnippet 1. Traditionele C# 2.0-code

In C# 3.0 en LINQ kan dit geschreven worden als:

```
string[] names = { «Peter», «Jan», «Wim», «Patrick», «Ann» };
var result = from n in names
             where n.Length == 3
             select n;
foreach( string s in result ) Console.WriteLine( s );
```

Codesnippet 2. Een voorbeeld van een query-instructie geschreven in C# 3.0

Hoe we tot deze eenvoudige syntax komen via de uitbreidingen in de C# 3.0-taal, lichten we nu stap voor stap toe.

Conditionele enumeratie in C# 2.0

Wanneer query-code wat complexer wordt, leiden foreach- en if-combinaties snel tot complexe sourcecode-constructies. De eenvoud kunnen we echter behouden door de conditie in de loop in te bouwen, wat we in de volgende pseudo-code beogen:

```
foreach ( string n in names.Where( n.Length == 3 ))
{
  Console.WriteLine(n);
}
```

Codesnippet 3. Pseudo-code

De pseudo-code uit snippet 3 kunnen we met weinig effort bouwen, omdat het foreach-keyword in feite staat voor een speciale enumeratie. Codesnippet 1 is namelijk gelijk aan:

```
IEnumerator ie = (names as IEnumerable).GetEnumerator();
while (ie.MoveNext())
{
  string n = (string) ie.Current;
  if (n.Length == 3)
    Console.WriteLine();
}
```

Codesnippet 4. Enumeratie-code in C# 2.0

Een enumerator in .NET is een object dat de IEnumerable-interface implementeert, het is een soort cursor die op iteratieve manier toegang verschaft tot een collectie. Die toegang tot de collectie kunnen we conditioneel beperken door onze eigen enumerator te definiëren die enkel elementen teruggeeft die aan onze conditie voldoen. C# 2.0 staat ons al toe om gebruik te maken van het Where-keyword dat een filteruitdrukking als parameter aanvaardt.

```
public delegate bool Filter<T>(T element);

public class MyStringCollection
{
    private string[] elements;

    public MyStringCollection(string[] elements)
    {
      this.elements = elements;
    }

    public IEnumerable<string> Where(Filter<string> f)
    {
      foreach (string element in elements)
          if (f(element) == true)
          yield return element;
    }
}
```

Codesnippet 5. Enumerator-definitie in C# 2.0 met Where-filter

In codesnippet 5 maken we hiervan gebruik, waardoor de class MyStringCollection een IEnumerable teruggeeft waarmee conditioneel door de collectie heen gelopen kan worden. Dit passen we toe in onze collectie om alleen strings met lengte 3 te selecteren; zie snippet 6:

```
MyStringCollection col = new MyStringCollection(names);
foreach (string n in col.Where(delegate(string s){return s.Length == 3;}))
{
```

```
    Console.WriteLine(n);
}
```

Codesnippet 6. Conditionele enumeratie in C# 2.0

Lambda-expressies in C# 3.0

Hoewel C# 2.0 al behoorlijk krachtig is, vraagt deze programmeertaal veel code om conditionele enumeraties te definiëren; we moeten met name een delegate-object aanmaken dat telkens de lengte van de string controleert. C# 3.0 maakt dit eenvoudiger met lambda-expressies, want een lambda-expressie is een verkorte manier om een delegate-object te definiëren. Daarbij gaat een lambda-expressie nog een stapje verder dan de anonieme methode.

Zo is bijvoorbeeld de anonieme methode:

```
delegate(string s) { return s.Length == 3; }
```

equivalent aan de lambda-expressie:

```
s => s.Length == 3
```

Een lambda-expressie is uitermate handig wanneer we filter-expressie nodig hebben, zoals in codesnippet 6, waar we het filter via een lambda-expressie vereenvoudigen tot:

```
MyStringCollection col = new MyStringCollection(names);
foreach (string n in col.Where( s => s.Length == 3 ))
{
    Console.WriteLine(n);
}
```

Codesnippet 7. Conditionele enumeratie m.b.v. lambda-expressies in C# 3.0

Een lambda-expressie is eigenlijk een instance van een delegate, hier 'Func' genaamd, met de volgende signatuur:

```
delegate R Func<T, R>(T element);
```

waarbij T het type is dat aan de linkerkant van het => symbool wordt gebruikt, en R het type dat aan de rechterkant wordt gebruikt. Als gevolg hiervan kan een delegate-variabele als volgt gedeclareerd worden:

```
Func<string, bool> filter = s => s.Length == 3;
```

Met behulp van deze Func-delegate gaan we nu de Where-functie uit codesnippet 5 aanpassen, zodat we een lambda-expressie als argument kunnen gebruiken:

```
public IEnumerable<string> Where(Func<string, bool> f)  {...}
```

We gaan nog een stap verder door de Where-functie op elke willekeurige collectie te gebruiken, dat we realiseren door middel van generics:

```
public static class Query
{
    public static IEnumerable<T> Where<T>(IEnumerable<T> col, Func<T, bool> f)
    {
        foreach (T element in col)
        {
            if (f(element) == true)
                yield return element;
        }
    }
}
```

Codesnippet 8. 'Where' is als statische functie bruikbaar op willekeurige collecties

In codesnippet 8 is 'Where' gedefinieerd als een statische functie waarmee een collectie van objecten van type T gefilterd kan worden. Hierdoor kunnen we het volgende schrijven:

```
foreach (string s in Query.Where<string>(names, s => s.Length == 3)) {...}
```

De aandachtige lezer ziet nog een laatste moeilijkheid: wie zou bovenstaande niet liever schrijven met behulp van een methode uit onze eigen collectie names, zoals gedefinieerd in snippet 2:

```
foreach ( string s in names.Where( s => s.Length == 3 ))   {...}
```

In specifieke gevallen lukt het wel om onze eigen collectie uit te breiden met een member-functie, maar meestal kunnen we niet zomaar elke collectie uitbreiden. Tenzij we gebruikmaken van de nieuwe techniek van extensiemethodes in C# 3.0.

Extensiemethodes (extension methods)

Met een extensiemethode kunnen we een methode toevoegen aan types zonder dat we die types hoeven te hercompileren. Bovendien laat de extensiemethode tegelijk specialisatie toe van deze methodes voor specifieke implementaties.
Stel dat we de String-class willen uitbreiden met een ToInt32-methode. Vroeger kon dit niet, maar met behulp van extensies kan dit. In codesnippet 9 definiëren we eerst een eigen statische class StringExtensions met statische ToInt32- methode:

```
public static class StringExtensions
{
    public static int ToInt32( this string s )
    {
        return int.Parse(s);
    }
}
```

Codesnippet 9. Voorbeeld van een extensiemethode om de String-class uit te breiden

Merk op dat een extensiemethode een statische methode is in een statische class. Omdat deze methode een ander type uitbreidt, moet het eerste argument van de extensiemethode het this keyword gebruiken. Als gevolg hiervan kunnen we met deze extensiemethode nu een string omzetten naar een int:

```
int i = "5".ToInt32();
```

Met deze techniek van extensiemethodes, kunnen we codesnippet 8 herschrijven:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> col, Func<T, bool> f)
{
    foreach (T element in col)
    {
        if (f(element) == true)
            yield return element;
    }
}
```

Codesnippet 10. Where-methode gedefinieerd m.b.v. een extensiemethode

Het this-argument wordt vervolgens door de compiler gebruikt om te bepalen waar de extensiemethode toegepast mag worden, in dit geval door IEnumerable<T>-types, zodat de volgende code mogelijk wordt:

```
foreach ( string s in names.Where( s => s.Length == 3 ))  {...}
```

De Where-functie heeft geen type meer nodig (dus geen Where<string>). Dit komt omdat de C# 3.0-compiler het type kan afleiden uit de code. Dit heeft als gevolg dat we eenvoudigere code kunnen schrijven. We hebben het met name over impliciet getypeerde variabele-declaraties.

Impliciet getypeerde variabele-declaraties

De C# 3.0-compiler kan dus het type bepalen van een variabele uit de context en hierdoor kunnen we een vereenvoudigde syntax gebruiken voor het declareren van variabelen.
Waar we in C# 2.0 dienen te schrijven:

```
List<Employee> employees = new List<Employee>();
```

...maken we in C# 3.0 gebruik van het nieuwe 'var'-keyword. Ook vragen we de compiler om het type van de variabele af te leiden uit het type van de expressie die we eraan toekennen:

```
var employees = new List<Employee>();
```

Object Initialization Expressions en Anonymous Types

Om elk element van een lijst met elementen om te zetten in een ander type kunnen we gebruikmaken van extensiemethodes. In codesnippet 11 loopt de extensiemethode Project over een lijst objecten van type T en converteert deze tot type K:

```
public static IEnumerable<K> Project<T, K>(
              this IEnumerable<T> col, Func<T, K> p)
{
   foreach (T element in col)
   {
     yield return p(element);
   }
}
```

Codesnippet 11. De extension-methode Project converteert collectie elementen tot type K

Als voorbeeld passen we de Project-extensiemethode toe op een lijst van Employee-objecten:

```
class Employee
{
   public Employee(string name) { this.name = name; }
   private string name;
   public string Name { get { return name; } set { name = value; } }
   private int age;
   public int Age { get { return age; } set { age = value; } }
}
```

Om die lijst van alle employees te maken, gebruiken we volgende code:

```
var employees = names.Where(n => n.Length == 3)
                .Project( empName => new Employee(empName));
foreach (Employee e in employees)
{
   ...
}
```

Codesnippet 12. De Project-methode converteert een employee-naam in een employee-object

Als de Employee-class geen constructors zou hebben (of geen enkele geschikte constructor), wordt het nog leuker, omdat we in C# 3.0 nu ook kunnen instantiëren met een zogeheten objectinitialisatie-expressie. Voor een Employee-instantie gebeurt dit als volgt:

```
new Employee { Name = empName , Age = 18}
```

Deze syntax lijkt sterk op de initialisatie van attributen en is equivalent aan:

```
Employee e = new Employee();
e.Name = empName;
e.Age = 18;
```

In gevallen waar onze Employee-class enkel gebruikt wordt als tussenresultaat, heeft onze class zelfs geen methodes nodig en laat C# 3.0 toe om een anoniem type te gebruiken:

```
...( new { Name = empName, Age = 18 } )

Toegepast op codesnippet 12 wordt dit:

var employees = names.Where(n => n.Length == 3)
                .Project( empName => new {Name = empName, Age = 18});
```

Codesnippet 13. Where selecteert, Project converteert

Het verschil tussen codesnippets 12 en 13 is dat we bij snippet 13 een collectie krijgen van objecten van anoniem type, en dat we daarom geen variabele kunnen declareren van een expliciet type. Hierdoor wordt het gebruik van het var-keyword essentieel.

Language Integrated Query (LINQ)

C# 3.0 voorziet ons van statements die wezenlijk vertaald worden naar extensiemethoden, lambda-expressies en dergelijke. Bekijk bijvoorbeeld de uitdrukking:

```
names.Where(n => n.Length == 3)
```

Deze uitdrukking kan met behulp van LINQ geschreven worden als

```
var result = from n in names
             where n.Length == 3
             select n;
```

Codesnippet 14. Voorbeeld van een LINQ-uitdrukking

Merk op dat het resultaat in codesnippet 14 een iterator oplevert en dus nog niet de uiteindelijke uitkomst bevat. Deze wordt pas berekend wanneer we het resultaat opvragen, bijvoorbeeld via een foreach-constructie:

```
foreach ( string s in result )  ...
```

Deze constructie is identiek aan een enumeratie, waarin eenvoudig voorgesteld de ie.MoveNext()-call de Where-extension aanroept:

```
IEnumerator<string> ie = result.GetEnumerator();
while( ie.MoveNext() ) ...
```

Dus pas als over het resultaat wordt geïtereerd, worden de extensiemethodes uitgevoerd om het resultaat te krijgen. Met deze uitgestelde uitvoering kan veel efficiënter worden omgegaan met geheugen, omdat op deze manier minder, of zelfs geen, tussenresultaten aangemaakt hoeven te worden.

Een complexer voorbeeld dat eveneens gebaseerd is op extensiemethodes is te zien in codesnippet 15.

```
from n in names
where n.Length == 3
orderby n
select new { Name = n, Length = n.Length };
```

Codesnippet 15. Deze LINQ-uitdrukking geeft alfabetisch gesorteerd een collectie terug van anoniem getypeerde objecten

Hierbij krijgen we een collectie terug van anoniem getypeerde objecten in alfabetische volgorde. De objecten zelf bevatten nu de naam en ook de lengte.

In een volgend voorbeeld zien we een join van twee collecties, waarbij we een collectie van vaders met kind terugkrijgen:

```
var pairs = from a in names, b in children
                  where b.Father.Name == a
                  select new { Name = a, Child = b };
```

Of we hebben een collectie van customers en we willen alle orders sinds 1998:

```
var orders = from c in customers,
                 o in c.Orders
             where o.OrderDate >= new DateTime(1998, 1, 1)
             select new {c.CustomerID, o.OrderID, o.OrderDate};
```

Of we willen alle customers sorteren op hun gemeente en naam:

```
var sortedCustomers = from c in customers
                      orderby c.City, c.Name
                      select c
```

We kunnen ook groeperen, bijvoorbeeld onze namen zoals te zien is in codesnippet 16.

```
var wordGroups = from n in names
                 group n by n[0] into g
                 select new {FirstLetter = g.Key, Words = g.Group};

foreach (var g in wordGroups)
{
  Console.WriteLine("Words that start with the letter '{0}':",
                    g.FirstLetter);
   foreach (var w in g.Words)
   {
     Console.WriteLine(w);
   }
}
```

Codesnippet 16. Deze LINQ-uitdrukking illustreert hoe gegroepeerd kan worden in een selectie

DLINQ en XLINQ

DLINQ en XLINQ zijn researchprojecten die de Language Integrated Query-feature van C# gebruiken waardoor het mogelijk is met dezelfde constructie selecties te maken uit databases en XML-infosets. Met bijvoorbeeld DLINQ kunnen we een .NET-type schrijven dat een tabel uit de database voorstelt:

```
[Table(Name="Customers")]
public class Customer {
   [Column]
   public string Name;
   [Column]
   public int Age;
   [Column]
   public bool IsVip;
}
```

Codesnippet 17. Voorbeeld met DLINQ

Hiermee kunnen we dan een query in C# schrijven:

```
Table<Customer> customers = ...
Table<Orders> orders = ...

var query = from c in customers, o in orders
            where o.Customer == c.Name
            select new { c.Name, o.OrderID, o.Amount, c.Age };
```

Codesnippet 18. Voorbeeld met DLINQ in C#

DLINQ maakt gebruikt van het feit dat dit soort van statement intern als een objectstructuur wordt voorgesteld (van type Expression<T>) die door DLINQ omgezet kan worden naar een SQL-query:

```
SELECT [t0].[Age], [t1].[Amount], [t0].[Name], [t1].[OrderID]
FROM [Customers] AS [t0], [Orders] AS [t1]
WHERE [t1].[Customer] = [t0].[Name]
```

Besluit

In C# 2.0 zette Microsoft de trend in naar vereenvoudigde constructies, zoals het bouwen van collecties met generics, eenvoudiger enumeraties, anonymous methods, etc... In C# 3.0 is men hier nog een stapje verder gegaan, en kan je bestaande types uitbreiden met extentie-methodes, delegates zijn nog eenvoudiger dankzij lambda-expressies, en kunnen we nu queries schrijven in C# zonder daarvoor SQL te gebruiken. Als laatste dienen we ook te vermelden dat alles wat we hier zagen gebeurd zonder uitbreidingen in Intermediate Language, en dat dit alles ook mogelijk is met VB 9.0!

Peter Himschoot is Microsoft Regional Director voor België en Nederland en werkt als trainer en .NET architect bij het Belgische opleidingsbedrijf U2U (www.u2u.net).

Wim Uyttersprot is oprichter en director van U2U. U2U is gespecialiseerd in de Microsoft Visual Studio 2005 en SQL Server 2005 Training en Consultancy.

Nuttige internetadressen

Meer informatie over LINQ: http://msdn.microsoft.com/netframework/future/linq
Voorbeelden: http://msdn.microsoft.com/vcsharp/future/linqsamples/default.aspx
De C# 3.0 site: http://msdn.microsoft.com/vcsharp/future
Experimenteer met C# 3.0 en LINQ: http://download.microsoft.com/download/4/7/0/4703eba2-78c4-4b09-8912-69f6c38d3a56/linq%20preview.msi