Maurice de Beijer is een onafhankelijk softwareontwikkelaar

Visual Basic 9.0 belooft veel goeds

EEN OVERZICHT VAN DE NIEUWE FEATURES VOOR DE VOLGENDE VERSIE VAN VISUAL BASIC

Tijdens de PDC die afgelopen september in Los Angeles werd gehouden, is veel tijd besteed aan de uitbreidingen die voor Visual Basic 9.0 zijn gepland. In dit artikel zal ik hiervan een overzicht geven en enkele van de belangrijkste features verder uitdiepen.

Language Integrated Query, kortweg LINQ genoemd, is een van de belangrijkste en de meest in het oog springende uitbreiding die we kunnen verwachten. LINQ op zich is geen Visual Basic-specifieke uitbreiding aangezien het ook in C# beschikbaar is. LINQ is zelfs in elke .NET-taal te gebruiken, maar dan niet op dezelfde geïntegreerde manier als in Visual Basic of C#. De query comprehensions zijn hierbij het deel van de compiler dat een 'SQL-achtige' query begrijpt en in de benodigde API-aanroepen omzet. Deze query comprehensions maken alles veel transparanter en beter begrijpbaar, maar het uiteindelijke programma zoals dat door de compiler wordt gegenereerd is natuurlijk hetzelfde. Zowel het Visual Basic- als het C#-team hebben voor het uitbreiden van de compiler gekozen, maar dan wel ieder op een eigen manier. Bij de keuze voor de syntax heeft het Visual Basic-team zich laten leiden door productiviteit en gekozen voor het reeds bekende, namelijk SQL. Het C#-team gebruikt een nieuwe syntax die de werkelijke volgorde volgt waarin de delen worden uitgevoerd. Dit verschil in opvatting tussen het Visual Basic- en C#-team, waarbij 'productiviteit' tegenover 'technische correctheid' staat, vormt de rode draad van dit verhaal. Hoewel de query-taal in Visual Basic veel op SQL lijkt is het dat niet. Indien een filter wordt opgegeven van namen die met een 'B' beginnen gebruiken we niet: LastName like "B%" maar: LastName.StartsWith("B").

Wat is LINQ nu eigenlijk precies en wat betekent het voor een ontwikkelaar? Aangezien LINQ een zeer uitgebreid onderwerp is en als zodanig een onderwerp voor een ander artikel is, volgt hieronder slechts een beknopte samenvatting. LINQ geeft de ontwikkelaar de mogelijkheid queries te maken over elke IEnumerable-collectie, wat zoveel wil zeggen als filters, joins, berekende waardes en projecties. Simpel gezegd het soort dingen dat met SQL op een databasetabel gebeurt, maar dan op een IEnumerable-collectie.

```
Dim SmallCountries = _
   Select Country _
   From Country In Countries _
   Where Country.Population < 1000000
```

Codevoorbeeld 1.
Een eenvoudige LINQ-query.

```
Dim SmallCapitals = _
   Select City _
   From City in Cities, Country In Capitals _
   Where City.Country = Country.Name
   And Country.Population < 1000000
   Order By City.Name
```

Codevoorbeeld 2.
Een LINQ-query met join en sortering.

De query in codevoorbeeld 1 levert een set van alle landen op met minder dan 1 miljoen inwoners.

In dit eenvoudige voorbeeld is meteen een andere uitbreiding te zien en dat zijn impliciete types. Verderop meer informatie hierover. Natuurlijk zijn complexere queries ook mogelijk en kun je verschillende collecties met elkaar joinen en het resultaat sorteren. In feite is nagenoeg elke soort bewerking binnen een LINQ-query mogelijk die in een SQL-query ook mogelijk is; zie codevoorbeeld 2.
Natuurlijk zijn de twee belangrijkste bronnen van data naast IEnumerable-collecties, namelijk databases en XML, niet vergeten en wordt hier direct een specifieke uitbreiding voor meegeleverd. Voor databases is dit DLINQ en voor XML is dit XLINQ, waarbij je met beide direct in de data kan zoeken zonder dit eerst in een standaardcollectie te laden. Doordat dit twee specifieke implementaties van hetzelfde onderliggende mechanisme zijn, wordt het ook mogelijk een LINQ-query te maken waarbij een databasetabel met een XML-document en een IEnumerable-collectie wordt gejoined tot een resultaatset. Aan de codevoorbeelden 1 en 2 kan je ook niet zien of de collectie met landen direct uit een database komt of een collectie in geheugen is. Om hier achter te komen is het nodig om naar het type van de collecties te kijken.

XML-integratie

In het .NET Framework 3.0 wordt een aantal nieuwe classes voor het werken met XML geïntroduceerd, de XDocument, XElement, XAttribute en nog een aantal vergelijkbare classes. Deze nieuwe types maken het mogelijk individuele XML-elementen te maken zonder eerst een allesomvattend document te maken. Ook deze

```
Dim xmlPerson = _
   <Person ID="12345">
      <FirstName>Maurice</FirstName>
      <LastName>de Beijer</LastName>
   </Person>
```

Codevoorbeeld 3.
Een deep XML-expressie.

```
Dim cFirstName As String = "Maurice"
Dim cLastName As String = "de Beijer"
Dim xmlPerson = _
   <Person ID="12345">
      <FirstName><%= cFirstName %></FirstName>
      <LastName><%= cLastName %></LastName>
   </Person>
```

Codevoorbeeld 4.
Een XML-literal expressie met dataholes.

```
Dim CountriesWithCapital As XElement = _
  <Countries>
  <%= Select <Country Name=(Country.Name)
     Density=(Country.Population/Country.Area)>
     <Capital>
       <Name><%= City.Name %></Name>
       <Longitude><%= City.Longitude %></Longtitude>
       <Latitude><%= City.Latitude %></Latitude>
     </Capital>
     </Country> _
  From Country In Countries, City In Capitals _
  Where Country.Name = City.Country %>
  </Countries>
```

Codevoorbeeld 5.

Een LINQ-query die direct XML oplevert.

```
Dim cFirstName As String = "Maurice"
Dim cLastName As String = "de Beijer"
Dim xmlPerson = New XElement("Person", _
  New XAttribute("ID", "12345"), _
  New XElement("FirstName", cFirstName), _
  New XElement("LastName", cLastName))
```

Codevoorbeeld 6.

Een XML-variabele maar dan met behulp van de classes.

```
Dim population = 31719
Dim name = "Belize"
Dim area = 1.9
Dim country = New Country{ .Name = "Nederland", ...}
```

Codevoorbeeld 7.

Variabele met een impliciet type.

```
Dim population As Integer = 31719
Dim name As String = "Belize"
Dim area As Float = 1.9
Dim country As New Country{ .Name = "Nederland", ...}
```

Codevoorbeeld 8.

Dezelfde variabele als in codevoorbeeld 7 maar dan met een expliciet type.

```
For Each Dim Country In SmallCountries
  Console.WriteLine(Country.Name)
Next
```

Codevoorbeeld 9.

Een For Each-lus met impliciete variabele type.

```
Dim Capitals = New List(Of City){ _
  { .Name = "Antanarivo", _
    .Country = "Madagascar", _
    .Longitude = 47.4, _
    .Lattitude = -18.6 }, _
  { .Name = "Belmopan", _
    .Country = "Belize", _
    .Longitude = -88.5, _
    .Latitude = 17.1 }, _
  { .Name = "Monaco", _
    .Country = "Monaco", _
    .Longtitude = 7.2, _
    .Latitude = 43.7 }, _
  { .Name = "Palau", _
    .Country = "Koror", _
    .Longitude = 135, _
    .Latitude = 8 } _
  }
```

Codevoorbeeld 10.

Een list die in één keer gevuld wordt.

nieuwe types zijn onderdeel van het .NET Framework en worden volop door C# gebruikt. Waar het verschil tussen C# en Visual Basic echter groot wordt is bij 'Deep XML Support', iets dat alleen Visual Basic heeft. 'Deep XML Support' wil niets anders zeggen dan dat de Visual Basic-compiler directe XML-literals in de code begrijpt zonder dat hier allemaal XML-type declaraties om heen staan; zie bijvoorbeeld codevoorbeeld 3.

Om delen van de XML variabel te maken, kan men hier variabele expressies in opgeven, de zogenaamde dataholes. Ook hier heeft het Visual Basic-team er weer voor gekozen om iets bekends te hergebruiken. Deze dataholes worden dan ook aangeven met dezelfde notatie, `<%= Variable %>`, zoals in ASP al gebruikelijk was. In codevoorbeeld 4 is een XML-literal expressie te zien met dataholes.

Het combineren van XML-literals met dataholes en LINQ-queries levert een zeer krachtige en compacte manier op om een XML-document te maken; zie codevoorbeeld 5.

Achter de schermen wordt de code uit codevoorbeeld 4 omgezet naar de volgende identieke code zoals in codevoorbeeld 6 is te zien. Dit laatste komt dan ook neer op de manier waarop dezelfde XML in C# gemaakt moet worden.

In een klein voorbeeld zoals dit is het al duidelijk dat de schrijfwijze met de XML-literals zoals in codevoorbeeld 4 veel duidelijker leesbaar is dan die in codevoorbeeld 6. Naarmate deze XML groter wordt, begint dit verschil steeds merkbaarder te worden. Tijdens zijn presentatie op de PDC maakte Paul Vick van het Visual Basic-team dit inzichtelijk. Hij had een logfile met data van de hits op zijn weblog en wilde hiervan een overzichtelijke spreadsheet maken. Met behulp van LINQ werden de data eerst gefilterd tot de juiste subset, maar er moest nog een spreadsheet worden gemaakt. Aangezien Excel standaard een spreadsheet als XML op kan slaan, heeft hij met Excel een geformatteerde template gemaakt en dit als XML bewaard. Nu kon hij dit XML-document even in Notepad openen en in zijn geheel in de Visual Basic-broncode kopiëren. Even een declaratie met 'Dim xlsDocument = ' er voor en de code kon gewoon weer compileren. Daarna nog even de voorbeeldregel uit de spreadsheet vervangen met het resultaat van de LINQ-query, iets dat zo gebeurd is met de nieuwe dataholes, het document bewaren en klaar is kees. Als je ditzelfde in C# wil doen, moet je elk XML-element of attribuut eerst van een New XElement of New XAttribute voorzien en dan ben je nog wel even bezig.

Impliciete variabele types

In Visual Basic 9 wordt het mogelijk om het type bij een variabele declaratie weg te laten en hier gelijk een waarde aan toe te kennen. Als dit gebeurt, zal de compiler zelf kijken wat voor type de waarde is en dit type voor de variabele gebruiken. Dit moet niet verward worden met late binding waarbij een variabele gedeclareerd wordt van het type Object waarna alle aanroepen hierop late bound zijn. Zie codevoorbeeld 7 voor een voorbeeld van impliciete variabelen.

Bij impliciete types wordt de variabele wel degelijk van een specifiek type, werkt IntelliSense gewoon en controleert de compiler het gebruik van de variabele. De volgende twee voorbeelden (7 en 8) laten het gebruik van impliciete variabele types zien en de vergelijkbare code zoals die nu geschreven wordt.

Naast variabelen die met Dim worden gedeclareerd is het ook mogelijk om de variabelen die in een lus, bijvoorbeeld een For Each- of een For Next-lus, gedeclareerd worden op deze manier van een type te voorzien; zie codevoorbeeld 9.

Type initialisatie

In codevoorbeelden 7 en 8 is gelijk een andere nieuwe mogelijkheid binnen Visual Basic 9.0 te zien en wel die van type-initi-

alisatie. Op de regel waar het object 'Country' gemaakt wordt, staat direct dat de 'Name'-property met 'Nederland' gevuld moet worden. Hier wordt niet gebruik gemaakt van een constructor met een Name-parameter, maar wordt van het nieuwe object ogenblikkelijk een property-waarde gezet. De lijst van properties die gezet moet worden staat tussen accolades, vergelijkbaar met de manier waarop een array geïnitialiseerd kan worden. Indien je meer waardes wil opgeven, dan kan dit door ze met een komma te scheiden. Behalve enkele objecten, zoals in het voorgaande voorbeeld is te zien, is het ook mogelijk een collectie in een keer te vullen met behulp van dezelfde syntax. Deze constructie is mogelijk met elke collectie die een Add()-functie ondersteunt; zie codevoorbeeld 10.

Relaxed delegates

In de huidige versie van Visual Basic is er een verschil tussen het aanroepen van een functie en het gebruiken van een event. Bij een functieaanroep bepaalt de compiler op basis van de parameters welke functie aangeroepen moet worden. Indien de parameters van een functiedefinitie niet exact hetzelfde zijn, maar van een basistype, bijvoorbeeld `Object` in plaats van `EventArgs`, dan wordt deze gewoon aangeroepen. In het geval van een event is dit echter niet het geval en moeten de parametertypes exact overeenkomen. Met de nieuwe Relaxed Delegates is dit niet langer een vereiste en kunnen de parameters gewoon van het type object zijn. Codevoorbeeld 11 toont twee event handlers die exact hetzelfde doen.

Een extra toevoeging hierop is dat je de parameters helemaal weg mag laten. Dit is gedaan omdat de parameters toch al vaak genegeerd worden en dus niets toevoegen. In codevoorbeeld 12 staat een nog compactere manier van het declareren.

Nullable types

Nullable types op zich zijn niet helemaal nieuw in Visual Basic 9 omdat deze in de huidige versie van Visual Basic 2005 al zijn geïntroduceerd. In de volgende versie gaat de ondersteuning echter een stuk verder. Zo worden de operators overloaded om met `Nothing` om te gaan. Indien één van de operands in een expressie `Nothing` is, zal het resultaat ook `Nothing` zijn; zie codevoorbeeld 13

```
Sub EventHandler(ByVal sender As Object, ByVal e As EventArgs) _
   Handles cmdButton.Click
   MessageBox.Show(cmdButton.Text)
End Sub

Sub RelaxedEventHandler(ByVal sender As Object, ByVal e As Object) _
   Handles cmdButton.Click
   MessageBox.Show(cmdButton.Text)
End Sub
```

Codevoorbeeld 11.
Twee event handlers die exact hetzelfde doen.

```
Sub VeryRelaxedEventHandler() Handles cmdButton.Click
   MessageBox.Show(cmdButton.Text)
End Sub
```

Codevoorbeeld 12.
Een nog compactere manier van het declareren van een event handler.

```
Dim A As Integer? = Nothing
Dim B As Integer? = 4711
Dim C As Integer? = A+B REM C = Nothing
```

Codevoorbeeld 13.
Werken met nullable variabele.

```
Dim A As Integer?
Dim B As Nullable(Of Integer)
```

Codevoorbeeld 14.
Declareren van nullable varabelen.

Behalve de operatoren is ook de declaratie van nullable variabelen uitgebreid en kan nu op een vergelijkbare en compacte manier als in C# gebeuren. De twee declaraties in codevoorbeeld 14 zijn exact hetzelfde.

Dynamic interfaces

Eén van de voordelen van late binding is dat de compiler design time niets over de objecten hoeft te weten. Je definieert gewoon een variabele van het type `Object`, zet Option Strict uit en je mag elke functie aanroepen. Dit voordeel heeft echter ook een groot nadeel, namelijk het ontbreken van IntelliSense, zodat de ontwikkelaar totaal geen hulp heeft bij het ontwikkelen. Om deze tekortkoming op te lossen, zonder het voordeel van late binding verloren te laten gaan, heeft men Dynamic Interfaces bedacht. Bij het gebruik van Dynamic Interfaces weet Visual Studio de structuur van het object en komt er dus IntelliSense, maar dwingt de compiler niets af en zorgt hij er voor dat alle aanroepen nog steeds late bound blijven. Codevoorbeeld 15 toont de twee mogelijkheden.

Dynamic identifiers

Alsof we nog niet dynamisch genoeg kunnen zijn bij het schrijven van code wordt ook nog een systeem van dynamische identifiers toegevoegd. Met dynamische identifiers wordt bedoeld dat zelfs de property-naam pas tijdens runtime bekend is. In codevoorbeeld 16 wordt het thuisadres gelezen. Door de variabele fieldName te veranderen van 'ThuisAdres' naar 'WerkAdres' wordt in plaats van het thuisadres het werkadres gelezen; zie codevoorbeeld 16. Dit is dus een wel heel extreme vorm van late binding.

```
<Dynamic()> _
Interface ISomeObject
   Property Name() As String
End Interface

Sub DynamicInterfaces()
   Dim name As String
   Dim someObject As Object
   name = someObject.Name

   Dim anotherObject As ISomeObject
   name = anotherObject.Name
End Sub
```

Codevoorbeeld 15.
Een late bound-object zonder IntelliSense-ondersteuning en een tweede met IntelliSense-ondersteuning.

```
Dim fieldName As String
fieldName = "ThuisAdres"
Dim person As New Person() { _
   .ThuisAdres = "Hier woon ik", _
   .WerkAdres = "Hier werk ik"}
Dim adres As String
adres = person.(fieldName)
```

Codevoorbeeld 16.
Pas tijdens runtime wordt bepaald welk field gelezen wordt.

```
<System.Runtime.CompilerServices.Extension> _
Module MyExtensions
   <System.Runtime.CompilerServices.Extension> _
   Function Count(Of T)([Me] As IEnumerable(Of T)) As Integer
      For Each Dim It In [Me]
         Count += 1
      Next
   End Function
End Module
```

Codevoorbeeld 17.
Een extension method die het mogelijk maakt om van elke IEnumerable-collectie een Count op te vragen.

Extension methods

Als we gaan kijken wat er bij LINQ werkelijk achter de schermen gebeurt, dan blijkt dat er een Where-functie aangeroepen wordt op de collectie. Omdat dit op elke IEnumerable-collectie mag gebeuren - en deze interface de Where-functie niet ondersteunt - is hier wat extra magie voor nodig in de vorm van extension methods. Extension methods maken het mogelijk extra functionaliteit toe te voegen aan al bestaande types zonder dat de originele types hierbij moeten worden uitgebreid of zelfs de broncode aanwezig is. Welke extension methods voor een bepaald type beschikbaar zijn, wordt bepaald door de `Imports`-statements die bovenaan een specifiek stuk code staan en de declaratie van de extension method zelf. Codevoorbeeld 17. toont het gebruik van een extension method waarmee je van elke IEnumerable-collectie een Count kunt opvragen.

Het maken van de extension methods zelf is voornamelijk bedoeld voor library- en framework-ontwikkelaars. Hiervoor is geen specifieke taaluitbreiding in Visual Basic gekomen om dit te ondersteunen, maar je kunt dit doen door gebruik te maken van de benodigde attributen.

Pre-alfa

Visual Basic 9 belooft weer veel goeds. Het Visual Basic-team is volop bezig om de productiviteit van individuele ontwikkelaars zo veel mogelijk te verhogen. Dit artikel is wel gebaseerd op een hele vroege, pre-alfa publicatie van de nieuwe features, dus het is goed mogelijk dat er nog dingen in veranderen. Inmiddels is technology preview van Visual Basic 9.0 LINQ beschikbaar op de Microsoft website http://msdn.microsoft.com/vbasic/future

Maurice de Beijer (www.TheProblemSolver.nl) is een MVP, onafhankelijk softwareontwikkelaar, en bètatester. Hij heeft zich gespecialiseerd in Visual Basic .NET, OOP, Visual FoxPro en het oplossen van moeilijke technische problemen. Maurice is de Problem Solver.

Referenties

Visual Basic 9.0 – msdn.microsoft.com/vbasic/future/default.aspx
Overview van Visual Basic 9.0 - msdn.microsoft.com/library/en-us/dnvs05/html/vb9overview.asp
The LINQ Project - msdn.microsoft.com/netframework/future/linq
Using Visual Basic as a Dynamic Programming Language - channel9.msdn.com/showpost.aspx?postid=116702
Future Directions for Data Programming in Visual Basic - channel9.msdn.com/showpost.aspx?postid=116700
The LINQ Project - msdn.microsoft.com/netframework/future/linq