Dennis van der Stelt

C# 3.0-features in .NET Framework 2.0

OVERSTAPPEN IS LONEND

Elk jaar een nieuw .NET Framework, elk jaar nieuwe technologieën. Wanneer houdt het op en waar wil Microsoft heen? Wellicht vragen die jij jezelf onlangs ook hebt gesteld. Dit artikel zal daar geen antwoord op geven, maar wel aantonen dat nieuwe features uit het .NET Framework 3.5 niet alleen voorbehouden zijn aan C# 3.0- of Visual Basic 9-projecten, maar ook bruikbaar zijn in jouw huidige .NET 2.0-projecten.

In het verleden betekende overstappen van .NET Framework 1.0 naar 1.1 of naar 2.0 daadwerkelijk een behoorlijke stap. Werkt het systeem nog wel naar verwachting en wat doen we met alle waarschuwingen van de conversiewizard? En nu komt Microsoft alweer met een nieuwe versie van zijn framework, inclusief een nieuwe versie van Visual Studio. LINQ ziet er interessant uit, maar welke investeringen moeten jij en jouw projectteam doen om de overstap te maken? Zoals mijn collega Anko Duizer in .NET Magazine #19 al heeft aangegeven, bestaat er in Visual Studio 2008 het zogeheten multi-targeting. Daarmee kun je .NET 2.0-, .NET 3.0- en .NET 3.5-projecten starten, met als resultaat dat je niet per ongeluk features van een later framework gebruikt. Maar is het daadwerkelijk zo dat we in onze .NET 2.0-projecten geen .NET 3.5-features kunnen gebruiken?

Het is allereerst belangrijk te weten dat het .NET Framework 3.5 nog steeds gebruikmaakt van de .NET 2.0-runtime, waar jouw applicatie daadwerkelijk in draait en de intermediate code vertaald wordt naar native code. Deze runtime is niet gewijzigd sinds .NET Framework 2.0, op bugfixes en performanceverbeteringen na. De verschillen staan in tabel 1. Dit betekent theoretisch dat al jouw .NET Framework 2.0-applicaties zonder enkel probleem in het nieuwe .NET Framework 3.5 draaien. Waarom draaien jouw .NET 3.5-applicaties dan niet net zo eenvoudig op .NET 2.0? De vraag is, wat maakt het verschil? Het antwoord is eigenlijk heel eenvoudig, namelijk de base-class libraries (BCL) van het .NET Framework en de compiler. Ik zal dit uitleggen door een aantal nieuwe features aan te halen. Daaruit zal blijken dat het wel degelijk mogelijk is C# 3.0 te gebruiken in een applicatie die alleen gebruikmaakt van het .NET Framework 2.0. Als de overstap naar het nieuwe .NET Framework 3.5 niet mogelijk is of nog op zich laat wachten, dan hoef je niet alle features uit C# 3.0 links te laten liggen.

Automatic properties in .NET 2.0

Wellicht ken je de code-snippet om eenvoudig een property aan te maken, geïntroduceerd in Visual Studio 2005. Je typt 'prop' en na een druk op de tabtoets verschijnt de code voor een property, inclusief een backing field (private member variable). Een nieuwe feature in .NET 3.5 is 'automatically implemented properties'. Er is genoeg te lezen over deze feature, maar waar het op neer komt is dat je met minder code hetzelfde resultaat bereikt. De .NET-run-time is echter niet gewijzigd, dus zit het verschil ergens anders. Voor deze feature is dat enkel de compiler. Deze maakt een volledig uitgeschreven property, inclusief een backing field. Met Reflector van Lutz Roeder kunnen we dit controleren. Dan is het dus ook heel eenvoudig te concluderen waarom je in Visual Studio 2008 eigenlijk de oude property-snippet niet meer nodig hebt. Ook bij het compileren van jouw .NET 2.0-projecten wordt de automatic property vertaald naar een property die herkenbaar is voor de .NET CLR 2.0. Dit is een duidelijk voorbeeld van een .NET 3.5-feature die ook in .NET 2.0-projecten bruikbaar is, aangezien het niet meer is dan 'syntactic sugar' in jouw programmeertaal, die door de compiler wordt herkend en uitgewerkt.

Extension methods in .NET 2.0

Een feature uit .NET Framework 3.5 zijn extension methods, waar je objecten kunt uitbreiden met een zelfgeschreven methode. Bijvoorbeeld een methode om een string te converteren naar leet-speak, waar je enkele karakters door cijfers wijzigt, zodat het 1337-sp3ak wordt; zie codevoorbeeld 1. In dit voorbeeld zie je dat we direct op het speak-object, de

	2002	2003	2005	2006	2007
Tool	Visual Studio .NET 2002	Visual Studio .NET 2003	Visual Studio 2005	Visual Studio 2005 + Extensies	Visual Studio 2008
C#	C# v1.0	C# v1.1	C# v2.0	zelfde versie	C# v3.0
Visual Basic	Visual Basic .NET v7.0	Visual Basic .NET v7.1	Visual Basic 2005 v8.0	zelfde versie	Visual Basic v9.0
Framework Libraries	NetFx v1.0	NetFx v1.1	NetFx v2.0	NetFx v3.0	NetFx v3.5
Engine (CLR)	CLR v1.0	CLR v1.1	CLR v2.0	zelfde versie	zelfde versie

Tabel 1. Versienummering .NET Framework

```
// Code uit je eigen applicatie
string speak = "Show the world I'm elite";
string result = speak.ToLeet();
Console.WriteLine(result);

// Hieronder de 'oude' manier van aanroepen
result = Extensions.ToLeet(speak);

// Extension method op een string
public static class Extensions
{
   public static string ToLeet(this string s)
   {
      s = s.Replace('l', '1');
      s = s.Replace('e', '3');
      return s.Replace('t', '7');
   }
}
```

Codevoorbeeld 1. Extension methods

string, onze extension method kunnen aanroepen. Wederom 'syntactic sugar', want de compiler maakt er code van, zoals te zien is in de regel code volgens de 'oude' manier.
Als je deze code in Visual Studio 2008 in een .NET 2.0-project wilt compileren, krijg je de melding dat de extension method niet gebruikt kan worden, omdat het ExtensionAttribute niet is te vinden. We kunnen met Reflector door de .NET Framework 3.5-assemblies naar de ExtensionAttribute-class zoeken. Dit is een voorbeeld van een class die in de BCL van .NET Framework 3.5 te vinden is en daarom niet direct te gebruiken is in .NET 2.0. Kijken we echter verder, dan zie je dat deze class een volledig lege class is en eigenlijk alleen als marker gebruikt wordt. Als je zelf een class met deze naam toevoegt aan je eigen project (zie codevoorbeeld 2 en let op de namespace), krijg je geen compiler warning meer en kun je extension methods in .NET 2.0-projecten gebruiken.

Object-initializers

Het is gebruikelijk om op een class een constructor te definiëren, waarmee we direct enkele waarden kunnen toekennen tijdens het instantiëren van het object. Biedt een class deze constructor niet, dan kunnen we de waarden veelal zetten via properties. Met object-initializers krijgen we echter een nieuwe mogelijkheid deze te zetten. In codevoorbeeld 3 zie je hoe een collectie van het type Training wordt aangemaakt en waarachter meteen het object wordt geïnitialiseerd. Dit wordt gedaan met collection-initializers, en meteen worden twee objecten Training aangemaakt, die worden gevuld via object-initializers. Je krijgt exact hetzelfde resultaat als we deze objecten per stuk aanmaken en met de properties de waarden toekennen, zoals je direct eronder kunt zien. Ook dit is 'syntactic sugar', waarbij code door de compiler gegenereerd wordt. Dit is een feature die nodig was om lambda- expressions te kunnen realiseren, maar als dit je voorkeur heeft, kun je voortaan op deze manier objecten en collecties initialiseren.

```
namespace System.Runtime.CompilerServices
{
   [AttributeUsage(AttributeTargets.Method |
        AttributeTargets.Class |
        AttributeTargets.Assembly)]
   public sealed class ExtensionAttribute : Attribute
   {
   }
}
```

Codevoorbeeld 2. ExtensionAttribute

```
var trainingen = new List<Training> {
   new Training {
      Titel = "LINQ",
      Dagen = 2 },
   new Training {
      Titel = "Mobile Development",
      Dagen = 2 }
};

var trainingen = new List<Training>();
Training training = new Training();
training.Titel = "LINQ";
training.Dagen = 2;
trainingen.Add(training);
```

Codevoorbeeld 3. Object-initializers

Local variable type inference

In codevoorbeeld 3 is ook te zien dat het var-keyword wordt gebruikt. Met LINQ kunnen we realiseren dat we een type (class) uit een query terug krijgen die niet bestaat. De compiler genereert dan zelf een class, maar omdat deze in Visual Studio nog geen naam heeft, wordt het var-keyword gebruikt. Dat de compiler zelf ontdekt wat voor type we nodig hebben, noemen we 'type inference'. Er wordt een 'anonymous type' gegenereerd. Anoniem omdat we tijdens het schrijven het type niet daadwerkelijk hebben gedefinieerd, maar enkel aangegeven hebben welke properties dit nodig heeft. Nog leuker wordt het als we van een anonymous type een hele collectie willen maken. Stel je voor dat we de Training-class niet hebben, maar wel over een lijst met trainingen willen beschikken. Dit kan door een extra methode te introduceren, zoals te zien is in codevoorbeeld 4. Bij het toevoegen van additionele (anonieme) trainingen, zijn we wel intellisense kwijt. Dit gaat zelfs Visual Studio iets te ver, hoewel de compiler het wel accepteert. Er is een mogelijkheid om type inference te gebruiken met collection-initializers om eenvoudig een collectie van anonieme types te creëren, maar de properties uit deze anonieme typen zijn niet meer te benaderen met intellisense of static typing en kunnen enkel met reflectie benaderd worden. Dat gaat dus zelfs de compiler een stap te ver.

LINQ en query-expression

In de meeste voorbeelden van LINQ zie je queries waar gebruik wordt gemaakt van query-expressions. Deze kun je herkennen aan de keywords from, where, select, enzovoort. Deze lijken het meest op een SQL-query. Ook dit is echter

```
static void Main()
{
   // Maak anoniem type en maak er een List<??> van.
   var training = new { Titel = "LINQ", Dagen = 2 };
   var trainingList = MakeList(training);
   // Voeg nieuw item toe aan anonieme lijst.
   trainingList.Add(new { Titel = "Mobile Development", Dagen = 2});
   trainingList.Add(new { Titel = "C# 3.5", Dagen = 4 });
}

public static List<T> MakeList<T>(params T[] items)
{
   List<T> newList = new List<T>();
   foreach (T item in items)
      newList.Add(item);
   return newList;
}
```

Codevoorbeeld 4. Type inference

```
public static bool KnownMethod(char c)
{
  return c >= 'A' && c <= 'Z';
}

public delegate bool MyDelegate(char c);

public static void PrintCharacters(string s, MyDelegate filter)
{
  foreach (Char c in s.ToCharArray())
  {
    if (filter(c)) Console.Write(c);
  }
  Console.WriteLine();
}

static void Main()
{
  PrintCharacters("Deze Urgente Data Encrypten", new
    MyDelegate(KnownMethod));

  PrintCharacters("C# Lijd Aan Softe Software Architecten",
    delegate(char c) { return c >= 'A' && c <= 'Z'; });

  PrintCharacters("Raad Of Cyprus Kan Schaatsen", c => c >= 'A' && c
    <= 'Z');
}
```

Codevoorbeeld 5. Van delegates naar lambda-expressions

'syntactic sugar', omdat deze vertaald worden naar de bijbehorende operators. Dit zijn vaak extension methods op de IEnumerable-interface waarachter de code ligt die gebruikmaakt van selectors, predicates, enzovoort, om te kunnen 'queryen' over de collecties. Deze queries kun je herkennen doordat de where-, select- en andere methodes achter elkaar worden geschreven in plaats van bij de query-expressions, waar elk keyword los wordt geschreven. Deze operators zijn een van de grote uitbreidingen in de base-class library van het .NET Framework 3.5. Hierdoor is deze feature niet te gebruiken in .NET Framework 2.0. Hadden we deze wel, dan nog zijn er andere nieuwe features van .NET Framework 3.5 die we nodig hebben om LINQ volledig toegankelijk te maken voor .NET 2.0.

Lambda-expressions

De laatste feature die ik beschrijf, zijn lambda-expressions. Deze bestaan al langer in functionele programmeertalen. Ze werden al in 1936 geïntroduceerd in de wiskunde en zijn dus niets nieuws onder de zon. Het gaat te ver om deze expressies tot in detail te behandelen, maar ze worden wel gezien als 'anonymous methods on steroids'. Anonymous methods zijn geïntroduceerd in .NET 2.0, maar hebben enkele limieten die lambda-expressions niet kennen.

- Lambda-expressies kunnen van de argumenten (parameters) het type zelf uitvinden dan wel ontdekken.
- Lambda-expressies kunnen zowel statement- als expression-codeblokken omvatten. Anonymous methods alleen statement-blokken.
- Lambda-expressies kunnen ook als argument gebruikt worden, waarna ze weer deelnemen in 'type inference' en 'method overload resolution'.
- Lambda-expressies met een expression-codeblok kunnen worden omgezet in expression-trees.

Ook lambda-expressions worden door de compiler geconverteerd naar code die door de .NET 2.0-runtime kan worden opgepakt. Om te zien hoe C# is geëvolueerd vanuit delegates via anonymous methods naar lambda-expression, kun je naar codevoorbeeld 5 kijken. In de main methode is op de eerste regel gebruikgemaakt van een normale delegate. In de tweede regel is gebruikgemaakt van een anonymous method (een nieuwe feature sinds .NET 2.0) en in de laatste regel een lambda-expression. Functioneel exact hetzelfde, alleen de meegegeven tekst is elke keer verschillend. Probeer eens zonder de computer er achter te komen wat er nu op het scherm komt.

In de voorbeelden op de tweede en derde regel zijn de delegate en de methode *KnownMethod* niet meer nodig. Maar zeker de lambda-expression is kort geschreven, hoewel deze initieel niet altijd even leesbaar overkomt. Leer ze echter gebruiken en je merkt na verloop van tijd hoe krachtig ze gebruikt kunnen worden.

Geen grote migratie

Zoals aangetoond bestaan enkele nieuwe features uit 'syntactic sugar'. De overige zijn aanvullingen op de base-class library van .NET. Er is geen nieuwe runtime gekomen en daardoor is er geen grote migratie nodig om over te gaan naar .NET 3.5. Niet voor jou als ontwikkelaar, maar ook niet voor jouw .NET 2.0-projecten. Daarnaast biedt Visual Studio 2008 zoveel nieuws, dat de overstap zeer zeker lonend is en er eigenlijk geen redenen zijn om met deze overstap te wachten.

Dennis van der Stelt is werkzaam als trainer/coach bij Class-A. Daarvoor heeft hij acht jaar gewerkt als softwareontwikkelaar. Sinds 2001 is Dennis bezig met .NET. Zijn speciale interesse gaat uit naar architectuur en gedistribueerde applicaties, Windows Communication Foundation, Mobile Development, Agile-principles & practices en alles wat nieuw is. Het blog van Dennis is te lezen op http://bloggingabout.net/blogs/dennis/. Voor vragen is hij bereikbaar via dennis.vanderstelt@class-a.nl.