# LINQ voor dagelijks gebruik

## INTEGREREN BESPAART VEEL TIJD

**Patrick Vorgers**

Met de introductie van het .NET framework 3.5 is ook versie 3.0 van C# geïntroduceerd. De uitbreidingen die in versie 3.0 zijn geïntroduceerd zijn geïnspireerd op functionele talen zoals Haskel en ML en zijn voornamelijk geïntroduceerd om Language Integrated Query of kortweg LINQ mogelijk te maken. In combinatie met extensiemethoden is het mogelijk om data op een uniforme manier in code te benaderen. Daardoor is het niet alleen voor databases te gebruiken maar ook voor interne datastructuren.

Met de introductie van elke nieuwe versie van het .NET framework heeft Microsoft de ontwikkelaar nieuwe gereedschappen gegeven om nog sneller applicaties te kunnen ontwikkelen. De laatste grote stap voor veel ontwikkelaars is toch wel de introductie van LINQ. Het biedt de ontwikkelaar mogelijkheden om data, in bijvoorbeeld de vorm van databases, collecties of XML op een eenvoudige manier te manipuleren.

### LINQ in meerdere gedaantes

Als een ontwikkelaar voor het eerst in aanraking komt met LINQ dan is het eerste dat opvalt de SQL-achtige syntax (zie codevoorbeeld 1). Daarom wordt het ook direct gekoppeld aan deze syntax. Dit is erg jammer aangezien LINQ veel meer te bieden heeft en zich niet beperkt tot deze manier van datamanipulatie.

```
List<Customer> customers = new List<Customer>
{
new Customer(1, "Peter", "Book") ,
new Customer(2, "Jane", "Shelf"),
new Customer(3, "Jake", "Shelf")
};

// Zoek de customer met customerid 1
var customerOnID =
from customer in customers
where (customer.CustomerID == 1)
select customer;
```

**CODEVOORBEELD 1: LINQ QUERY**

Om dit goed te kunnen uitleggen is het belangrijk om iets meer te weten over de werking en implementatie van LINQ. Het is gebaseerd op de Common Language Runtime (CLR) 2.0. Dit houdt in dat met de introductie van LINQ er geen aanpassingen nodig waren in de bestaande CLR en dat alle nieuw geïntroduceerde C# taalconstructies direct vertaald konden worden naar CLR 2.0 instructies. Dit betekent ook dat het mogelijk moet zijn een LINQ query zoals in Codevoorbeeld 1 te herschrijven in C# 2.0 stijl, zonder gebruik te maken van LINQ-taalconstructies. Codevoorbeeld 2 is hier een voorbeeld van en is gelijk aan Codevoorbeeld 1. Nu wordt alleen gebruik gemaakt van C# 2.0 taalconstructies.

```
// C# 2.0 stijl
IEnumerable<Customer> customerOnID =
customers.Where<Customer>(delegate(Customer customer)
{
return customer.CustomerID == 1;
});

</code>
```

**CODEVOORBEELD 2: LINQ QUERY IN C# 2.0 STIJL**

Het eerste dat opvalt is de 'Where'-methode op de Customers-collectie. Deze methode is een extensiemethode. Deze methodes kunnen achteraf aan een klasse worden toegevoegd. De LINQ is gebaseerd op het gebruik van extensiemethodes. Het voegt namelijk aan de generieke IEnumerable interface veel methodes toe die specifiek voor LINQ zijn. Doordat LINQ de generieke IEnumerable uitbreidt wordt het mogelijk om LINQ-functionaliteit te gebruiken op elke klasse die deze interface implementeert. De 'Where'-methode is één van die methodes. Daarmee is het bijvoorbeeld mogelijk om op basis van een conditie elementen uit de collectie te selecteren. Dit is vergelijkbaar met de werking van de 'Where'clause in SQL.

```
// Lambda expression
IEnumerable<Customer> customerOnID =
customers.Where(x => x.CustomerID == 1);
```

**CODEVOORBEELD 3: LINQ QUERY MET BEHULP VAN EEN LAMBDA EXPRESSIE**

In plaats van gebruik te maken van anonymous methods zoals in Codevoorbeeld 2, is het ook mogelijk om LINQ te gebruiken in combinatie met lambda expressies (zie Codevoorbeeld 3). Deze code is nog steeds gelijk aan Codevoorbeeld 1 en 2, maar maakt gebruik van een andere notatiewijze. Dat maakt het vooral voor ontwikkelaars eenvoudiger om LINQ-functionaliteit op een korte, krachtige manier toe te passen. De rest van het artikel maakt gebruik van deze notatiewijze en laat zien dat de LINQ-extensiemethoden op plaatsen kunnen worden gebruikt waar dit in eerste instantie niet voor de hand ligt.

## Functionaliteiten met LINQ-extensie

Voordat er kan worden gestart met de verschillende voorbeelden moet er eerst nog even worden gekeken naar de Customer-klasse (zie Codevoorbeeld 4). Deze klasse wordt in bijna elk voorbeeld gebruikt en verdient dus een kleine toelichting. Als er in een voorbeeld wordt gerefereerd naar een Customer dan wordt deze klasse bedoeld. De Customer-klasse is ten behoeve van de voorbeelden volledig uitgekleed en bevat alleen een aantal properties en geen methoden (exclusief de constructors). Van de drie properties die zijn gedefinieerd behoeft wellicht alleen de CustomerID property uitleg. Met behulp van deze property kan een Customer uniek worden geïdentificeerd en is dus uniek over alle Customer-objecten.

```
public class Customer
{
  public int CustomerID { get; set; }
  public string FirstName { get; set; }
  public string LastName { get; set; }

  public Customer(int customerID, string firstName, string lastname)
  {
this.CustomerID = customerID;
this.FirstName = firstName;
this.LastName = lastname;
      }
  public Customer(int customerID)
    : this(customerID, string.Empty, string.Empty)
  {
  }
}
```

**CODEVOORBEELD 4: CUSTOMER KLASSE**

## Functionaliteit 1: Zoeken in collecties

Collecties, als ontwikkelaars kennen we ze allemaal en hebben er allemaal mee te maken, want wat is software zonder data. Het opslaan van data in deze collecties is maar de helft van de functionaliteit die we nodig hebben aangezien de data er ook weer uitgehaald moet kunnen worden. Veelal worden er dan 'Find'-methodes geschreven die de data op basis van een unieke identificatie terug kunnen vinden. Deze 'Find'-methodes zijn star en moeten worden aangepast en/of uitgebreid als we de data op een andere manier willen ophalen of in plaats van één element meerdere elementen willen ophalen. Het wordt helemaal complex als er moet worden gezocht in complexe datastructuren die bestaan uit een hiërarchie van collecties.

```
List<Customer> customers = new List<Customer>
         {
            new Customer(1, "Peter", "Book") ,
            new Customer(2, "Jane", "Shelf"),
            new Customer(3, "Jake", "Shelf")
         };
// Zoek de customer met customerid 1
Customer customerOnID =
  customers.FirstOrDefault(x => x.CustomerID == 1);
// Zoek de eerste customer met als voornaam Peter
Customer customerOnFirstName =
  customers.FirstOrDefault(x => x.FirstName == "Peter");
 // Zoek alle customers met als achternaam Shelf
 List<Customer> customersShelf =
  customers.Where(x => x.LastName == "Shelf").ToList();
```

**CODEVOORBEELD 5: ZOEKEN IN COLLECTIES**

Met behulp van LINQ query syntax (zie codevoorbeeld 1) is het mogelijk om eenvoudig een query te maken die het gevraagde resultaat oplevert. Door echter rechtstreeks gebruik te maken van de extensiemethodes in combinatie met lambda-expressies is het mogelijk om veel gerichter tot het gewenste resultaat te komen (zie Codevoorbeeld 5).

Het zoeken in collecties is eigenlijk de basisfunctionaliteit van LINQ en ligt erg voor de hand en de meeste ontwikkelaars zullen ook wel op het idee komen om het hiervoor te gebruiken. Wellicht niet door de extensiemethoden, maar zeker wel door de query syntax. Het volgende voorbeeld is echter een heel ander verhaal aangezien hierin de extensiemethodes worden gebruikt om een array te initialiseren.

## Functionaliteit 2: Array initialisatie

Bij het declareren van een array moet deze vaak ook worden geïnitialiseerd. Normaal gesproken wordt dit gedaan door middel van een array declaratie waarna met behulp van een for-loop elk element uit deze array wordt geïnitialiseerd met de juiste waarde (zie Codevoorbeeld 6).

```
// Maak een array van 10 Customer elementen
Customer[] array = new Customer[10];
for (int index = 0; index < array.Length; index++)
  array[index] = new Customer(index);

// Maak een array van 10 Customer elementen aan (1, 2, 3 .. 10)
Customer[] arrayCustomerRange1 =
  Enumerable.Range(1, 10).Select(x => new Customer(x)).ToArray();
// Maak een array van 10 Customer elementen aan (10, 20, 30 .. 100)
Customer[] arrayCustomerRange2 =
  Enumerable.Range(1, 10).Select(x => new Customer(x * 10)).
  ToArray();
// Maak een array van 10 verschillende customer elementen aan met
id 1
Customer[] arrayCustomerRepeat1 =
  Enumerable.Repeat(1, 10).Select(x => new Customer(x)).ToArray();
// Maak een array van 10 dezelfde customer elementen aan met id 1
Customer[] arrayCustomerRepeat2 =
  Enumerable.Repeat(new Customer(1), 10).ToArray();
```

**CODEVOORBEELD 6: ARRAY INTIALISATIE**

Met behulp van LINQ kan deze initialisatie echter eenvoudiger worden opgeschreven door gebruik te maken van de 'Repeat' en de 'Range' extensiemethoden. Voor grote arrays is het ondanks de elegantie van de LINQ oplossing niet aan te raden om deze te gebruiken aangezien LINQ de array dynamisch laat groeien. Hierdoor worden er veel tijdelijke arrays aangemaakt die later weer door de garbage collector moeten worden opgeruimd. Dit is te vergelijken met het gebruik van de StringBuilder klasse in plaats van de + operator om strings samen te voegen.

## Functionaliteit 3: Het casten van collecties

Voor LINQ was het niet mogelijk om in C# of VB een sequence of collectie van het type U te casten naar een sequence of collectie van het type T, ondanks dat U afgeleid was van het type T. Het is dus bijvoorbeeld niet mogelijk om een 'List<Customer>' te casten naar een 'List<object>' ondanks dat 'Customer' en alle andere klasses in .NET als basisklasse 'object' hebben. Met behulp van LINQ kan dit wel worden gerealiseerd met behulp van de extensiemethode 'Cast'.

```
ArrayList customers = new ArrayList
{
new Customer(1, "Peter", "Book"),
new Customer(2, "Jane", "Shelf")
};
// Cast een niet generieke enumerable (ArrayList) naar een gene-
rieke enumerable (Customer)
IEnumerable<Customer> customersSeq1 = customers.Cast<Customer>();
List<Customer> customersList1 = customers.Cast<Customer>().
ToList();
```

```
List<object> objectsList = new List<object>
{
new Customer(1, "Peter", "Book"),
new Customer(2, "Jane", "Shelf")
};
// Cast een generieke enumerable van het type U (object) naar een
ander type (Customer)
IEnumerable<Customer> customersSeq2 = objectsList.
Cast<Customer>();
List<Customer> customersList2 = objectsList.Cast<Customer>().To-
List();
```

**CODEVOORBEELD 7: CASTEN VAN ENUMERABLES**

Met de 'Cast' extensiemethode is het tevens mogelijk om een niet generieke collectie te converteren naar een getypeerde generieke collectie. Het voordeel hiervan mag duidelijk zijn. De compiler heeft nu tijdens het compileren al de mogelijkheid om te controleren of er tijdens het gebruik van de collectie altijd met het juiste type wordt gewerkt. In beide varianten gaat de 'Cast' extensiemethode er vanuit dat er in de aangeboden sequence objecten zitten van hetzelfde type en dat dit type afleid is van het type waar naar toe wordt gecast. Als dit niet het geval is, dan kun in plaats van de extensiemethode 'Cast' de extensiemethode 'OfType' worden gebruikt. Deze extensiemethode levert een sequence op met daarin alleen de objecten van het gevraagde type.

## Functionaliteit 4: Collectie van lengte 1

Op het eerste gezicht lijkt dit een stukje functionaliteit dat niet vaak wordt gebruikt, maar in de praktijk blijkt dit toch in twee situaties vaak voor te komen. In het eerste geval wordt een methode aangeroepen die een collectie van elementen als één van de inputparameters verwacht, terwijl aan de aanroepende kant we slechts te maken hebben met één element. In het andere geval wordt deze constructie gebruikt bij het retourneren van een collectie als het resultaat van een functie waarbij het resultaat uit slechts één element bestaat.
In het codevoorbeeld wordt ervan uitgegaan dat de eerste situatie waarbij een methode als inputparameter een collectie verwacht. Om deze methode te kunnen aanroepen moet het element waar op dat moment mee wordt gewerkt worden gewrapped in een array/collectie, zodat deze kan worden meegegeven.

```
// Maak een customer aan met id 1
Customer customerValue = new Customer(1, "Peter", "Book");

// Maak een array aan van lengte 1 en initialiseer deze
Customer[] array = new Customer[1];
array[0] = customerValue;
CustomMethod(array);

// Maak een sequence aan van lengte 1
IEnumerable<Customer> sequence = Enumerable.Repeat(customerValue,
1);
CustomMethod(sequence);
```

**CODEVOORBEELD 8: COLLECTIE VAN LENGTE 1**

Met behulp van de extensiemethode 'Repeat' kan dit eenvoudig worden herschreven. De extensiemethode 'Repeat' maakt het mogelijk om een sequence van een bepaalde lengte te creëren dus ook van lengte 1. Deze sequence kan dan op zijn beurt weer worden gebruikt in de aanroep van de methode. Hierdoor hoeft er geen wrapper object te worden gecreëerd die de IEnumerable interface implementeert.

## Functionaliteit 5: ForEach/ForAll

De reden van het Microsoft C# team om wel de 'ForEach'-methode op de generieke List te implementeren, maar niet op alle andere collecties zal wel altijd een raadsel blijven. Echter met behulp van extensiemethodes is het eenvoudig om deze functionaliteit zelf te realiseren. Aangezien LINQ gebruik maakt van de generieke IEnumerable interface is het verstandig om hierbij aan te sluiten en ook de extensiemethode 'ForAll' hierop te implementeren.

```
public static void ForAll<T>(this IEnumerable<T> source,
 Action<T> action)
{
if (source == null)
      throw new ArgumentNullException("source");
if (action == null)
      throw new ArgumentNullException("action");

// Loop all the elements and perform the action
     foreach (T item in source)
      action(item);
}

// ForEach List
List<Customer> customersList = new List<Customer>
{
new Customer(1, "Peter", "Book"),
new Customer(2, "Jane", "Shelf")
};
customersList.ForEach(x => Console.WriteLine("Customer {0} {1}", x.
FirstName, x.LastName));

// ForAll Dictionary
Dictionary<int, Customer> customersDict = new Dictionary<int, Cus-
tomer>
{
{1, new Customer(1, "Peter", "Book")},
{2, new Customer(2, "Jane", "Shelf")}
};
customersDict.ForAll(x => Console.WriteLine("Customer {0} {1}", x.
Value.FirstName, x.Value.LastName));
```

**CODEVOORBEELD 9: FOREACH/FORALL**

De definitie van de 'ForAll'-extensiemethode zorgt ervoor dat alle klasses die de generieke IEnumeable interface implementeren ook gebruik kunnen maken van de 'ForAll' methode. Ook de generieke List klasse die al de 'ForEach' implementeert.

## Functionaliteit 6: Meerdere loops combineren

In sommige gevallen is het noodzakelijk om een operatie op meerdere sequences uit te voeren. In plaats van gebruik te maken van een foreach loop op elke sequeqnce, is het met behulp van de LINQ-extensiemethode 'Concat' mogelijk om de sequences te combineren. Hierdoor kan door middel van één foreach loop over alle items in één keer worden geïtereerd.

```
Customer peterBook = new Customer(1, "Peter", "Book");
Customer janeShelf = new Customer(2, "Jane", "Shelf");
Customer jakeShelf = new Customer(3, "Jake", "Shelf");
List<Customer> oldCustomers =
  new List<Customer> { peterBook, janeShelf };
List<Customer> newCustomers =
  new List<Customer> { jakeShelf };

// Lijsten combineren
foreach (Customer customer in oldCustomers.Concat(newCustomers))
  Console.WriteLine("Customer {0} {1}", customer.FirstName,
  customer.LastName);
```

**CODEVOORBEELD 10: MEERDERE LOOPS COMBINEREN**

Codevoorbeeld 5 is een voorbeeld van zo'n gecombineerde iteratie. In dit voorbeeld wordt een lijst van bestaande klanten gecombineerd met een lijst van nieuwe klanten. De 'Concat'-extensiemethode werkt op de generieke IEnumerable interface, dus het is niet noodzakelijk dat de collecties die worden gecombineerd ook van hetzelfde type moeten zijn. Hierdoor is het mogelijk om

array's met lists te combineren zolang deze maar de generieke IEnumerable interface implementeren.

## Functionaliteit 7: Stringmanipulatie

Aangezien de 'string' klasse ook de IEnumerable interface implementeert, is het ook mogelijk om alle LINQ extensiemethoden op strings toe te passen. Op het eerste gezicht lijkt dit niet veel toe te voegen aan de functionaliteit die de 'string' klasse standaard al biedt.

```
string tekst1 = "Microsoft .NET magazine";
string tekst2 = "artikel";

// Substring (.NET)
tekst1.Skip(10).Take(4).AsString();
// Verwijder karakters (Microsoft NET magazine)
tekst1.Where(x => x != '.').AsString();
// Join van 2 strings (irtaaie)
tekst1.Join(tekst2, t1 => t1, t2 => t2, (t1, t2) => t1).AsString();
```

**CODEVOORBEELD 11: STRINGMANIPULATIE**

De eerste twee voorbeelden in Codevoorbeeld 11 zijn eenvoudiger op te lossen door gebruik te maken van de 'Substring' en 'Replace'-methoden. Echter het 'joinen' van twee strings is een heel ander verhaal. Met de 'Join'-methode is het mogelijk om een string te maken van karakters die zowel in de eerste als in de tweede string zitten. Een ander voorbeeld is het gebruik van set operaties zoals 'Except', 'Intersect', 'Union' en 'Distinct' (zie set operaties). Het is inderdaad de vraag of het gebruik van LINQ op strings erg veel praktische toepassingen heeft. Het laat echter wel zien hoe krachtig LINQ is, doordat het kan worden gebruikt op alle klassen die de IEnumerable implementeren.

## Functionaliteit 8: Oneindige reeksen

Codevoorbeeld 12 is niet specifiek voor LINQ maar toont aan hoe kort, krachtig en als aller belangrijkste begrijpelijk een operatie op een oneindige reeks kan worden opgeschreven in LINQ. De definitie van de oneindige reeks maakt gebruik van de 'yield return' om elke keer een nieuw kwadraat terug te geven wanneer daar om wordt gevraagd.

```
public static IEnumerable<int> Squares
{
  get
  {
    int value = 0;
    // Maak een oneindige loop aan om de kwadraten te berekenen
    while (true)
    {
      // Bereken het kwadraat en geef deze terug
      yield return (value * value);
      value++;
    }
  }
}

// Neem de kwadraten van 3 t/m 7 en druk deze af
Integers.Squares.Skip(3).Take(5).ForAll(x => Console.
WriteLine("Kwadraat: {0}", x));
// Dit werkt niet aangezien de reeks oneindig is
Integers.Squares.Reverse().Take(5).ForAll(x => Console.
WriteLine("Kwadraat: {0}", x));
```

**CODEVOORBEELD 12: REEKS VAN KWADRATEN**

In het eerste voorbeeld worden de eerste drie kwadraten overgeslagen en daarna wordt er een collectie van vijf kwadraten aangemaakt. Het tweede voorbeeld daarentegen gaat eerst proberen een sequence te maken van de oneindige lijst en dan van de laatste vijf. Dit is natuurlijk niet mogelijk, omdat er in dit voorbeeld wordt gewerkt met een oneindige reeks. Het geeft wel duidelijk aan dat bij het gebruik van extensiemethoden er altijd na moet worden gedacht over het feit of deze de gehele sequence nodig heeft om zijn operatie uit te kunnen voeren.

## Functionaliteit 9: Set operaties

C# biedt de ontwikkelaar een groot scala taalconstructies die het leven als ontwikkelaar een stuk eenvoudiger maken. Maar het is in C# nog steeds niet mogelijk om gebruik te maken van sets en daar operaties op uit te voeren. Met de introductie van LINQ klopt dit statement niet meer. Met LINQ is het nu mogelijk om setoperaties zoals bijvoorbeeld 'Intersect' en 'Union' uit te voeren op een willekeurige collectie zolang deze maar de generieke IEnumerable interface implementeert.

```
Customer peterBook = new Customer(1, "Peter", "Book");
Customer janeShelf = new Customer(2, "Jane", "Shelf");
Customer jakeShelf = new Customer(3, "Jake", "Shelf");
List<Customer> customers1 =
  new List<Customer> { peterBook, peterBook, janeShelf };
List<Customer> customers2 =
  new List<Customer> { jakeShelf, peterBook };

// Intersect (Peter Book)
customers1.Intersect(customers2);
// Union (Peter Book, Jane Shelf, Jake Shelf)
customers1.Union(customers2);
// Distinct (Peter Book, Jane Shelf)
customers1.Distinct();
// Except (Jane Shelf)
customers1.Except(customers2);
```

**CODEVOORBEELD 13: SET OPERATIES**

De setoperaties maken gebruik van de standaard vergelijkingsmethoden zoals deze in .Net voorhanden zijn. Dit houdt in dat objecten alleen aan elkaar gelijk zijn wanneer het ook daadwerkelijk dezelfde objecten zijn. Dit is niet altijd wenselijk. Om dit probleem op te vangen, is het ook mogelijk om een eigen comparer mee te geven aan de setoperaties.
Het is wel jammer dat het niet mogelijk is om door middel van standaard operatoren (& | == !=) de set operaties aan te geven. Dit zou de leesbaarheid en het gebruik van sets zeker ten goede komen.

## Conclusie

Een aantal van deze voorbeelden zullen niet direct hun toepassing terugvinden in productiesoftware. Maar ze laten echter wel goed zien wat de krachtige mogelijkheden van LINQ zijn. De basis van LINQ ligt bij de functionele talen. De extensiemethodes laten dit ook duidelijk zien. Veel van deze methoden vinden dan ook hun oorsprong in deze talen. Vooral de functionele kijk op problemen vergt voor veel ontwikkelaars een verandering in denken. Het is iets waar hij even aan moet wennen en wat hij zichzelf stapje voor stapje moet eigen maken. Maar eenmaal onder de knie dan heeft hij een krachtig stuk gereedschap in handen dat zeker thuis hoort in de bagage van elke .NET ontwikkelaar.

**Links**

The LINQ project - http://msdn.microsoft.com/en-us/netframework/aa904594.aspx
.NET Framework - http://en.wikipedia.org/wiki/.NET_Framework
C# - http://en.wikipedia.org/wiki/C_Sharp%28Programming_language%29
Generic type parameter variance in the CLR - http://blogs.msdn.com/rmbyers/archive/2005/02/16/375079.aspx

**Patrick Vorgers,** is als technisch architect werkzaam bij de Management en Consultancy afdeling van Ordina Software Integration & Development (www.ordina.nl). Zijn specialisatie is software architecturen, high availability en software engineering. Hij is te bereiken op patrick.vorgers@ordina.nl.


# www.politie-ict.nl

**Werk:** omvangrijke en complexe projecten, innovatieve werkomgeving, beveiliging cruciaal, grote organisatie, veel verantwoordelijkheid.

**Privé:** economische zekerheid, locatie in de regio, mogelijkheid 4x9 uur werken, opleidingsmogelijkheden, aantrekkelijke spaar- en verlofregelingen.

POLITIE

vts Politie Nederland