LINQ for Objekter med C#

LINQ er den store nye teknologi med .NET 3.5. Der er i virkeligheden tale om en familie af teknologier som deler et fælles fundament - LINQ for Objekter. Artiklen gennemgår bl.a. alle de mange LINQ operatorer med eksempler.

Skrevet den **06. Feb 2009** af **nielle** I kategorien **Programmering / C#** | ★★★☆☆

Indledning

LINQ er en forkortelse af *Language INtegrated Query*. Altså noget med at skrive forespørgelser eller søgninger (querys) direkte som en del af programmeringssprogets syntaks. Det er indført med den nyligt udgivne .NET 3.5.

Sproget er her f.eks. C# eller VB.NET. Artiklens eksempler er skrevet i C#, men syntaksen ligner meget den du finder i VB.NET.

I virkeligheden er LINQ en familie af teknologier som baserer sig på et fælles fundament: LINQ for Objekter, LINQ for DataSet, LINQ for SQL og LINQ for XML. Det er endda muligt at udvide med sin egen LINQ for Noget - hvis man har en passende "Noget" at gøre det for :^)

LINQ for Objekter, som denne artikel altså handler om, er på mange måder lig med det omtalte fælles fundament. De andre LINQ teknologier tilføjer så "blot" nye elementer til fundamentet. En stor del af fundamentet består af de såkaldte *LINQ operatorer* og de vil derfor dominere artiklens sidste 3/4-del.

Helt basalt set drejer LINQ sig om at løbe igennem, søge i og generelt at manipulere med sekvenser af "et eller andet". I LINQ for Objekter handler det om arrays og f.eks. i LINQ for SQL handler det om rækker fra en eller flere tabeller i en database.

I det følgende vil jeg bruge ordet *sekvens* om "et eller andet" som har en indbygget naturlig rækkefølge. Det kunne være et array som gennemløbes et element ad gangen, det kunne være en streng som gennemløbes bogstav for bogstav, en tabel i et DataSet, en List<type>, en ArrayList, en tabel i en database eller noget helt syvende.

v. 1.0: 26/02/2008 - Første version.

Davs LINQ

Lad mig lægge ud med et eksempel:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace LinqToObjects
{
    class Program
    {
```

```
        static void Main(string[] args)
        {
            string[] landArr = { "Danmark", "Sverige", "Norge", "Island",
"Finland", "Tyskland" };

            IEnumerable<string> landQuery = landArr
                .Where(l => l.EndsWith("land"))  // Kun lande som slutter med
"land"
                .OrderBy(l => l);  // Sorteret i stigende alfabetisk orden.

            foreach (string land in landQuery)
                Console.WriteLine(land);
        }
    }
}
```

Resultat: Finland, Island og Tyskland.

Den centrale stump kode er dette:

```
            IEnumerable<string> landQuery = landArr
                .Where(l => l.EndsWith("land"))
                .OrderBy(l => l);
```

Koden er delt over flere linjer for at gøre det mere tydeligt hvad der egentlig foregår (det er typisk at LINQ eksempler vises på den måde :^).

Samlet på én linje er det:

```
            IEnumerable<string> landQuery =
                landArr.Where(l => l.EndsWith("land")).OrderBy(l => l);
```

Selve skrivemåden kaldes for *extension method syntax* (eller *lambda query syntax*) fordi der er gjort brug af extension metoder og lambda udtryk. Men mere om det lige om lidt.

Alternativt kan der skrives i *query comprehension syntax*, som ser sådan ud:

```
            IEnumerable<string> landQuery = from l in landArr
                                            where l.EndsWith("land")  // Kun
lande som ...
                                            orderby l  // Sorteret i stigende
alfabetisk orden.
                                            select l;
```

Man kan på ingen måder sige at den ene syntaks er at foretrække frem for den anden. Extension method syntax er måske lidt voluminøs når eksemplerne bare bliver en smule mere kompliceret, men den har til gengæld også en hel del metoder, som ikke er til stede i query comprehension syntax. Omvendt er query comprehension syntax mere kompakt, generelt lettere at læse og man ser desuden tydeligt hvordan den er inspireret af SQL.

Heldigvis udelukker de ikke hinanden, og man kan endda kombinere dem i noget vi kunne kalde for mikset syntaks:

```
            int[] nogleTal = { 1, 7, 3, 9, 4, 7, 4, 8, 7, 43, 32, 9, 0, 0, 34,
2 };

            IEnumerable<int> query = (from t in nogleTal
                                      where t < 20
                                      orderby t
                                      select t).Distinct();

            foreach (int tal in query)
                Console.WriteLine(tal);
```

Resultat: 0, 1, 2, 3, 4, 7, 8 og 9.

Lambda udtryk

I Where()-leddet fra før, så vi denne dims:

l => l.EndsWith("land")

Dette kaldes for et *lambda udtryk* (engelsk: *lambda expression*) og er indført med C# 3.0. Den primære grund var faktisk netop at bane vejen for LINQ.

Hvordan skal dimsen læses? Where() forventer i dette tilfælde at man levere et string-objekt (navnet på et land) og at der returneres en boolsk værdi (om det ender på "land" eller ej). Man kan udspecificere dette ved at sige at den kræver en delegate med følgende signatur:

```
delegate bool EnderPåLandDelegate(string l);
```

Indførelsen af de anonyme metoder i C# 2.0 var egentlig blot et mellemstadie til indførelsen af lambda udtrykkene:

C# 1.0 - Navngivne metoder:

En delegate (f.eks. en event-handler) skulle i C# 1.0 assignes med en metode man havde oprettet et andet sted i koden:

```
EnderPåLandDelegate c1 =
```

```
    EnderPåLand;

static public bool EnderPåLand(string l) {
    return l.EndsWith("land");
}
```

Dette var lidt omstændigt - specielt hvis funktionen faktisk kun skulle bruges til det ene formål. Derfor de *anonyme udtryk* med C# 2.0:

C# 2.0 - Anonyme metoder:

Med indførelsen af anonyme metoder kunne man i stedet definere metoden i samme linje hvor den assignes til delegaten:

```
EnderPåLandDelegate c2 =
    delegate(string l) { return l.EndsWith("land"); };
```

C# 3.0 - Lambda udtryk:

Med lambda udtryk er syntaksen blevet gjort endnu mere kompakt:

```
EnderPåLandDelegate c3 =
    (string l) => l.EndsWith("land");
```

eller blot:

```
DanskSprogetFilmDelegate c3 =
    l => l.EndsWith("land");
```

idet C# slutter sig til typen af argumentet, l (for land), via sammenhængen som den bruges i.

Extension metoder

Extension metoder er en anden ny ting med .NET 3.5. De gør det muligt at udvide en eksisterende type med metoder som den ikke har i forvejen - og det helt uden at man behøver at lave den om.

De implementeres som statiske metoder i en statisk hjælpe-klasse. Her er et eksempel som laver et relativ simpelt primtalstjek:

```
    static class NumberTheory
    {
        public static bool IsPrime(this int tal)  // Læg mærke til "this"
        {
            if (tal < 2) return false;
```

```
            for (int divisor = 2; divisor <= Math.Sqrt(tal); divisor++)
                if (tal % divisor == 0) return false;

            return true;
        }
    }
```

Med denne på plads kan man skrive:

```
            int t = 3;
            Console.WriteLine("Tallet {0} er et primtal: {1}", t,
NumberTheory.IsPrime(t));
```

eller blot (og netop dét er så det nye):

```
            Console.WriteLine("Tallet {0} er et primtal: {1}", t,
t.IsPrime());
```

Og det er ikke en gang nødvendigt at angive typen siden at C# kan slutte sig til den ud fra sammenhængen:

```
            Console.WriteLine("Tallet {0} er et primtal: {1}", 4,
4.IsPrime());
```

Med denne på plads, kan vi filtrere på en talserie for primtal:

```
            int[] tal = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> primtal = tal
                .Where(t => t.IsPrime());

            foreach (int t in primtal)
                Console.WriteLine(t);
```

Resultat: 2, 3, 5 og 7.

En lidt mere avanceret måde at opnå det samme er ved at tilføje følgende til NumberTheory-klassen:

```
        public static IEnumerable<int> PrimeFilter(this IEnumerable<int>
talSerie)
        {
            foreach (int tal in talSerie)
                if (tal.IsPrime())
                    yield return tal;
        }
```

Vi kan så skrive:

```
            IEnumerable<int> primtal = tal
                .PrimeFilter();

            foreach (int t in primtal)
                Console.WriteLine(t);
```

LINQ operatorerne fra ovenfor, While() og OrderBy(), er i virkeligheden implementeret som extension metoder på nogenlunde denne facon.

LINQ operatorer

While() og OrderBy() er kun et par af de rimeligt mange *LINQ operatorer* der findes. I det følgende vil jeg beskrive de enkelte funktioner og hvordan man bruger dem.

Eksemplerne er primært i extension method syntax, men i de tilfælde hvor at det er muligt at gøre det tilsvarende i query comprehension syntax er denne også angivet.

Eksemplerne er med vilje holdt meget simple. Koden forsøger ikke at være genial eller blot snedig, men det skal den heller ikke være: Formålet er at demonstrere principperne.

Operatorerne er opdelt i grupper sammen med andre funktioner af samme grundlæggende type.

Restriktion / Filtrering

Operatorer: Where, Take, TakeWhile, Skip, SkipWhile og Distinct.

Disse operatorer udvælger de elementer fra sekvensen som fortsat skal være med.

Where(betingelse): Udvælger alle de elementer som opfylder betingelsen.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Where(l => l.Contains("an"));

            // Query comprehension syntax
```

```
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              where l.Contains("an")
                                              select l;
```

Resultat: Danmark, Island, Finland og Tyskland.

Take(n): Udvælger de n første elementer af sekvensen og skipper resten.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Take(2);

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat: Danmark og Sverige.

TakeWhile(betingelse): Tager elementerne indtil at betingelsen er falsk første gang. Skipper resten.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .TakeWhile(l => l.Length > 5);

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat Danmark og Sverige.

Skip(n): Springer over de første n elementer fra sekvensen, og beholder resten.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Skip(4);

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat: Finland og Tyskland.

SkipWhile(betingelse): Springer over sekvensens elementer indtil at betingelsen er falsk første gang.

```
            // Extension method syntax
```

```
            IEnumerable<string> landeQuery1 = landeArr
                .SkipWhile(l => l.Length > 5);

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat: Norge, Island, Finland og Tyskland.

Distinct(): Fjerner dubletter fra sekvensen.

```
            string[] landeArr = { "Danmark", "Sverige", "Norge", "Island",
                                  "Finland", "Tyskland", "Sverige",
"Island"};

            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Distinct();

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat: Danmark, Sverige, Norge, Island, Finland og Tyskland.

Projektering

Operatorer: Select og SelectMany.

Disse operatorer oversætter det fremsøgte fra en form til en anden.

Select(oversætter):

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Select(l => l.ToUpper());

            // Query comprehension syntax
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              select l.ToUpper();
```

Resultat: DANMARK, SVERIGE, NORGE, ISLAND, FINLAND og TYSKLAND.

Et udtryk skrevet i query comprehension syntax skal altid startes med et from-led og afsluttes med et select-led. Der er dog intet i vejen for at bruge den trivielle select:

```
```

```
                                            select l;
```

En Select() vil altid være af formen et-element-ind-et-element-ud.

SelectMany(oversætter):

Hvor Select() altså er 1-til-1 så er SelectMany() 1-til-mange.

```
            char[] splitter = { 'a' };

            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .SelectMany(l => l.Split(splitter));

            // Query comprehension syntax
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              from ls in l.Split(splitter)
                                              select ls;
```

Resultat: D, nm, rk, Sverige, Norge, Isl, nd, Finl, nd, Tyskl og nd.

Joining

Operatorer: Join og GroupJoin.

Før koden, må vi lige have noget passende data som vi kan joine på. Først et par structs som definerer hvordan data ser ud:

```
    struct PokerSpiller
    {
        public readonly int id;
        public readonly string forNavn;
        public readonly string efterNavn;

        public PokerSpiller(int id, string forNavn, string efterNavn)
        {
            this.id = id;
            this.forNavn = forNavn;
            this.efterNavn = efterNavn;
        }
    }

    struct PokerRunde
    {
        public readonly int spillerId;
        public readonly float gevinst;
```

```
        public PokerRunde(int spillerId, float gevinst)
        {
            this.spillerId = spillerId;
            this.gevinst = gevinst;
        }
    }
```

Derefter data for et "typisk" spil poker:

```
            PokerSpiller[] spillere = {
                                        new PokerSpiller(1, "James Butler",
"Hickok"),
                                        new PokerSpiller(2, "Jack",
"McCall"),
                                        new PokerSpiller(3, "Martha Jane",
"Cannary-Burke")
                                    };

            PokerRunde[] runder = {
                                        new PokerRunde(1, 200),
                                        new PokerRunde(2, 15),
                                        new PokerRunde(2, 135),
                                        new PokerRunde(1, 355),
                                        new PokerRunde(1, 502),
                                        new PokerRunde(13, 2)
                                    };
```

Læg mærke til at spilleren "Martha Jane" ikke har vundet noget, samt at der er en ukendt spiller med id 13. Der er i øvrigt heller ikke taget højde for at der er nogen som jo må have tabt.

Join(): Denne operator parrer de to sekvenser på de elementer der er med i begge sekvenser. Det er det der kaldes for en INNER JOIN i SQL.

```
            // Extension method syntax
            IEnumerable stat = spillere.Join(  // Første sekvens - "den indre
sekvens".
                runder,  // Anden sekvens - "den ydre sekvens".
                s => s.id, r => r.spillerId,  // Der joines på disse to
værdier.
                (s, r) => new { s.efterNavn, r.gevinst }  // Resultatet.
                );

            foreach (var post in stat)
                Console.WriteLine(post);
```

Resultat:

```
{ efterNavn = Hickok, gevinst = 200 }
{ efterNavn = Hickok, gevinst = 355 }
{ efterNavn = Hickok, gevinst = 502 }
{ efterNavn = McCall, gevinst = 15 }
{ efterNavn = McCall, gevinst = 135 }
```

Det lille "var" der sneg sig ind i eksemplet er et tilfælde af en *anonym variabel*. C# 3.0 tillader at man bruger var i stedet for at skulle definere klasser specielt til at tage imod mellemresultater. Var'en er dog stadig typestærk, og C# ved i dette tilfælde at den indeholder et string-felt som hedder "efterNavn" og et float-felt som hedder "gevinst".

Visse steder kan brugen af "var" simplificere koden og gøre den nemmere at læse. Generelt bør man dog forsøge at skrive "pæn" kode og ikke lade alt for meget være underforstået på denne måde. Var kan kun bruges for lokale variable - den kan f.eks. ikke bruges som returtype for en funktion.

Samme eksempel, men denne gang i query comprehension syntax:

```
            // Query comprehension syntax
            IEnumerable stat = from s in spillere
                               join r in runder on s.id equals r.spillerId
                               select new { s.efterNavn, r.gevinst };
```

Personligt synes jeg at denne form er nemmere at læse.

I modsætning til i SQL, hvor at det er ligegyldigt hvad der er på den ene og den anden side af "on" i join'en, så kan rækkefølgen ikke byttes om i LINQ. Det skal skrives på formen:

```
            IEnumerable query = from y in ydreSeq
                                join i in indreSeq on y.aaa equals i.bbb
                                select new { y.aaa, y.bbb, i.ccc };
```

Jeg har ikke helt gennemskuet hvorfor at man kalder det for hhv. *inner sequence* og *outer sequence* i LINQ. Men det er for så vidt også lige meget. Det vigtigste er at man er klar over at der skelnes skarpt mellem de sekvenser der er på den ene side og på den anden side af en join.

Martha Jane kommer slet ikke med og det er fordi at Join() fungere som en INNER JOIN - da hendes spillerId ikke optræder i listen over runder, bliver hun ikke nævnt.

Ønsker man at have hende med alligevel, skal man have fat i GroupJoin() - denne svare til en LEFT OUTER JOIN:

GroupJoin(): Denne operator parrer og gruppere de to sekvenser. For hvert element i den første sekvens (den ydre) dannes der en gruppe over de resultater i den anden sekvens (den indre).

```
            // Extension method syntax
            var stat = spillere.GroupJoin(  // Første sekvens - "den indre
sekvens".
                runder,  // Anden sekvens - "den ydre sekvens".
```

```
                s => s.id, r => r.spillerId,  // Der joines på disse to
værdier.
                (s, rundeGrp) => new { s.efterNavn, rundeGrp }  // Resultatet.
                );

            // Query comprehension syntax
            var stat = from s in spillere
                       join r in runder on s.id equals r.spillerId
                       into rundeGrp
                       select new { s.efterNavn, rundeGrp };

            // Udskrivning
            foreach (var post in stat)
            {
                Console.WriteLine(post.efterNavn);

                foreach (PokerRunde runde in post.rundeGrp)
                    Console.WriteLine("\t" + runde.gevinst);
            }
```

Resultat:

```
Hickok
    200
    355
    502
McCall
    15
    135
Cannary-Burke
```

Ordning / Sortering

Operatorer: OrderBy, OrderByDecending, ThenBy, ThenByDecending og Reverse.

Disse operatorer ordner sekvensen ifølge den valgte sorterings-funktion.

OrderBy(sortering): Sortere sekvensens elementer i stigende orden efter sorterings-funktionen.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .OrderBy(l => l.Length);

            // Query comprehension syntax
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              orderby l.Length
                                              select l;
```

Resultat: Norge, Island, Danmark, Sverige, Finland og Tyskland.

OrderByDecending(sortering): Sortere sekvensen i faldende orden efter sorterings-funktioen.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .OrderByDescending(l => l.Length);

            // Query comprehension syntax
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              orderby l.Length descending
                                              select l;
```

Resultat: Tyskland, Danmark, Sverige, Finland, Island og Norge.

Man kan vælge at tilføje et "ascending" i OrderBy() eksemplet. Imidlertid er det default opførelse at der skal sorteres i stigende orden så det er normalt ikke nødvendigt.

Af samme årsag er "decending" obligatorisk hvis sorteringen skal være faldende i stedet for stigende.

ThenBy(sekundær sortering): Denne skal altid bruges sammen med OrderBy() eller OrderByDecending(). Den angiver en sekundær sortering efter at der er sorteret for første gang:

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .OrderBy(l => l.Length)  // Sorter efter længde...
                .ThenBy(l => l);  // ...dernæst alfabetisk

            // Query comprehension syntax
            IEnumerable<string> landeQuery2 = from l in landeArr
                                              orderby l.Length, l
                                              select l;
```

Resultat: Norge, Island, Danmark, Finland, Sverige og Tyskland.

ThenByDecending(sekundær sortering): fungere på samme måde, blot efter faldende sortering.

Reverse(): Sortere sekvensen i modsat rækkefølge.

```
            // Extension method syntax
            IEnumerable<string> landeQuery1 = landeArr
                .Reverse();

            // Query comprehension syntax
            // ... findes ikke ...
```

Resultat: Tyskland, Finland, Island, Norge, Sverige og Danmark.

Gruppering

Operator: GroupBy (det er den eneste i denne gruppe - GroupJoin() hører ikke med her).

GroupBy(gruppering): Denne grupperer sekvensens elementer i 1 eller flere grupper; en gruppe for hver mulig værdi af grupperings-funktionen.

```
            // Extension method syntax
            IEnumerable<IGrouping<int, string>> landeQuery1 = landeArr
                .GroupBy(l => l.Length);

            // Query comprehension syntax
            IEnumerable<IGrouping<int, string>> landeQuery2 = from l in
landeArr
                                                              group l by
l.Length;

            foreach (IGrouping<int, string> group in landeQuery1)
            {
                Console.WriteLine("Lande med {0} bogstaver:", group.Key);

                foreach (string land in group)
                    Console.WriteLine("\t{0}", land);
            }
```

Resultat:

```
Lande med 7 bogstaver:
    Danmark
    Sverige
    Finland
Lande med 5 bogstaver:
    Norge
Lande med 6 bogstaver:
    Island
Lande med 8 bogstaver:
    Tyskland
```

Og i stigende rækkefølge:

```
            // Extension method syntax
            IEnumerable<IGrouping<int, string>> landeQuery1 = landeArr
                .OrderBy(l => l.Length)
                .GroupBy(l => l.Length);

            // Query comprehension syntax
            IEnumerable<IGrouping<int, string>> landeQuery2 = from l in
```

```
landeArr
                                                                orderby l.Length
                                                                group l by
l.Length;
```

Mængder (engelsk: set)

Operatorer: Concat, Union, Intersect og Except.

Disse operatorer tager to sekvenser og kombinere dem f.eks. til deres fællesmængde.

Concat(sekvens #2): Kombinere sekvens #1 med sekvens#2. Eventuelle dubletter beholdes.

```
            int[] primtal = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
            int[] fibonacci = { 1, 1, 2, 3, 5, 8, 13, 21 };

            IEnumerable<int> query = primtal.Concat(fibonacci);
            foreach (int tal in query)
                Console.WriteLine(tal);
```

Resultat: 2, 3, 5, 7, 11, 13, 17, 19, 23, 1, 1, 2, 3, 5, 8, 13 og 21.

Union(sekvens #2): Kombinere sekvens #1 og #2 til deres foreningsmængde.
Eventuelle dubletter smides væk.

```
            IEnumerable<int> query = primtal.Union(fibonacci);
            foreach (int tal in query)
                Console.WriteLine(tal);
```

Resultat: 2, 3, 5, 7, 11, 13, 17, 19, 23, 1, 8 og 21.

Intersect(sekvens #2): Tager fællesmængden af de to sekvenser. Beholder udelukkende de elementer som er med i begge sekvenser.

```
            IEnumerable<int> query = primtal.Intersect(fibonacci);
            foreach (int tal in query)
                Console.WriteLine(tal);
```

Resultat: 2, 3, 5 og 13.

Except(sekvens #2): Beholder de elementer i sekvens #1 som ikke er til stede i sekvens #2. Kaldes for deres komplementærmængde.

```
            IEnumerable<int> query = primtal.Except(fibonacci);
            foreach (int tal in query)
                Console.WriteLine(tal);
```

Resultat: 7, 11, 17, 19 og 23.

Man kunne mene at Empty() operatoren, som behandles til sidst i denne artikel, faktisk burde være inkluderet i blandt mængde-operatorerne. Men det er den traditionelt ikke..

Konvertering

Operatorer: OfType, Cast, ToArray, ToList, ToDictionary, ToLookup, AsEnumerable og AsQueryable.

Disse operatorer ændrer sekvensens elementer fra en type til en anden.

OfType<type>: Acceptere en ikke-typet sekvens og konvertere den til en typet sekvens. Hvis der er elementer som ikke kan castes til den ønskede type, smides de bort.

```
            ArrayList ugeneriskHeld = new ArrayList();
            ugeneriskHeld.Add(7);
            ugeneriskHeld.Add(9);
            ugeneriskHeld.Add(13);
            ugeneriskHeld.Add("Sort kat");  // Ignoreres senere.

            IEnumerable<int> generiskHeld = ugeneriskHeld.OfType<int>();
            foreach (int forHeld in generiskHeld)
                Console.WriteLine(forHeld);
```

Resultat: 7, 9 og 13.

Cast<type>: Som OfType() men kaster en exception ved det første element som ikke kan castes til den ønskede type.

```
            ArrayList ugeneriskHeld = new ArrayList();
            ugeneriskHeld.Add(7);
            ugeneriskHeld.Add(9);
            ugeneriskHeld.Add(13);
            ugeneriskHeld.Add("Sort kat");  // Resultere i en exception
senere.

            IEnumerable<int> generiskHeld = ugeneriskHeld.Cast<int>();
            foreach (int forHeld in generiskHeld)
                Console.WriteLine(forHeld);
```

Resultat: 7, 9, 13 og så en InvalidCastException.

ToArray<type>: Konvertere sekvensen til et array af den angivne type.

```
int[] arrayHeld = ugeneriskHeld
    .OfType<int>()
    .ToArray<int>();
foreach (int forHeld in arrayHeld)
    Console.WriteLine(forHeld);
```

ToList<type>: Som ToArray(), men returnere i stedet en liste.

```
List<int> listHeld = ugeneriskHeld
    .OfType<int>()
    .ToList<int>();
foreach (int forHeld in listHeld)
    Console.WriteLine(forHeld);
```

For beskrivelser af ToDictionary() og ToLookup() vil jeg henvise til dokumentationen. De bruges ikke direkte så tit og er måske nok mere beregnet til intern brug i LINQ motoren.

AsEnumerable og *AsQueryable*:
I eksemplerne i denne artikel er der brugt IEnumerable fordi at dette er det interface som definerer alle LINQ operatorerne. Bevæger man sig over i LINQ for SQL bruges der imidlertid IQueryable. Denne arver fra IEnumerable og overstyre mange af metoderne i denne - dvs. at LINQ operatorerne kan opføre sig en smule anderledes alt efter om sekvensen er af er den ene eller anden type. AsEnumerable og AsQueryable bruges til at caste frem og tilbage mellem de to - for dermed at tvinge LINQ motoren til at bruge den ene eller anden variant.

Element-operatorer

Operatorer: First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault, ElementAt, ElementAtOrDefault og DefaultIfEmpty.

Disse operatorer udvælger et enkelt element fra sekvensen.

Hvis der ikke er noget element som opfylder kriteriet vil der normalt blive smidt en exception. Dette kan undgås ved at vælge en af Default-varianterne.

Hvad der eksakt er "default" afhænger af typen. I det følgende er typen en System.Int32 (int) og der er default lig med 0. Hvis det f.eks. er en System.String ville default være null i stedet.

Hvis der er mere end ét element i sekvensen som opfylder udvælgelses kriteriet vil der også blive smidt en exception.

First(betingelse) og *FirstOrDefault(betingelse)*: Udvælger det første element fra sekvensen som opfylder betingelsen.

```
            int[] primtal = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };

            Console.WriteLine(primtal.First());  // 2
            Console.WriteLine(primtal.First(p => p >= 10));  // 11
            // Console.WriteLine(primtal.First(p => p >= 100));  - Kaster en
exception.
            Console.WriteLine(primtal.FirstOrDefault(p => p >= 100));  // 0
```

Last(betingelse) og *LastOrDefault(betingelse)*: Som First(), men tager i stedet det sidste element fra sekvensen som opfylder betingelsen.

```
            Console.WriteLine(primtal.Last());  // 23
            Console.WriteLine(primtal.Last(p => p <= 10));  // 7
            // Console.WriteLine(primtal.Last(p => p <= 1));  - Kaster en
exception.
            Console.WriteLine(primtal.LastOrDefault(p => p <= 1));  // 0
```

Single(betingelse) og *SingleOrDefault(betingelse)*: Fungere egentlig som First() og Last(), men kræver at der er præcist et element som opfylder betingelsen.

```
            Console.WriteLine(primtal.Single(p => p % 2 == 0));  // 2
            // Console.WriteLine(primtal.Single(p => 10 <= p && p <= 20)); -
Kaster en exception.
            // Console.WriteLine(primtal
                .SingleOrDefault(p => 10 <= p && p <= 20));  - Kaster en
exception.
            Console.WriteLine(primtal.SingleOrDefault(p => 10 <= p && p <=
12));  // 11
            Console.WriteLine(primtal.SingleOrDefault(p => 8 <= p && p <=
10));  // 0
```

ElementAt(int): Returnere elementet ved det angivne indeks.

```
            Console.WriteLine(primtal.ElementAt(5));  // 13
            // Console.WriteLine(primtal.ElementAt(10));  - Kaster en
exception.
            Console.WriteLine(primtal.ElementAtOrDefault(10));  // 0
```

DefaultIfEmpty: Denne bruges til at sikre at der altid returneres et eller andet, hvis man starter med den tomme sekvens. Den er speciel nyttig i de tilfælde hvor at et mellemresultat giver en tom værdi.

```
            // Til dem der ikke har vundet noget.
            PokerRunde defaultRunde = new PokerRunde(-1, 0);

            // Extension Method Syntax
            var stat = spillere.GroupJoin(  // Første sekvens - "den indre
sekvens".
                runder,  // Anden sekvens - "den ydre sekvens".
                s => s.id, r => r.spillerId,  // Der joines på disse to
værdier.
                (s, rundeGrp) => new { s.efterNavn, rundeGrp }  // Resultatet.
                );

            foreach (var post in stat)
            {
                Console.WriteLine(post.efterNavn);

                foreach (PokerRunde runde in
post.rundeGrp.DefaultIfEmpty(defaultRunde))
                    Console.WriteLine("\t" + runde.gevinst);
            }
```

Resultat:

```
Hickok
    200
    355
    502
McCall
    15
    135
Cannary-Burke
    0
```

Aggregerings-operatore

Operatorer: Count, LongCount, Min, Max, Sum, Average og Aggregate.

Disse operatorer virker kun for sekvenser bestående af tal. De kan f.eks. bruges til at returnere summen af alle elementerne, eller at finde det største element.

```
            int[] primtalShuffle = { 13, 23, 17, 3, 2, 11, 5, 7, 19 };

            Console.WriteLine(primtalShuffle.Count());  // 9 (System.Int32)
            Console.WriteLine(primtalShuffle.LongCount());  // 9
(System.Int64)
            Console.WriteLine(primtalShuffle.Count(p => p >= 13));  // 4

            Console.WriteLine(primtalShuffle.Min());  // 2
            Console.WriteLine(primtalShuffle.Max());  // 23
```

```
            Console.WriteLine(primtalShuffle.Max(p => 2 * p));  // 2*23 = 46

            Console.WriteLine(primtalShuffle.Sum());  // 100
            Console.WriteLine(primtalShuffle.Average());  // 11,111...

            Console.WriteLine(
                primtal
                .Aggregate(1, (produkt, p) => produkt * p)  // 1 *
13*23*17*...*19 = 223092870
                );  // Aggregate hed tidligere Fold
```

Quantifier-operatore

Operatorer: Contains, Any, All og SequenceEqual.

Disse operatorer bruges til at teste om en given sekvens opfylder et ønsket kriterium.

```
            Console.WriteLine(primtal.Contains(11));  // true - 11 er et at
tallene

            Console.WriteLine(primtal.Any(p => 10 < p && p < 12));  // true

            Console.WriteLine(primtal.All(p => p % 2 == 0));  // false - 2 er
lige

            Console.WriteLine(primtal.SequenceEqual(  // false
                primtalShuffle
                ));

            Console.WriteLine(primtal.SequenceEqual(  // true
                primtalShuffle.OrderBy(p => p)
                ));
```

Generator-operatorer

Operatorer: Empty, Range og Repeat.

Disse bruges til at oprette en sekvens.

```
            // Empty - opretter en tom sekvens:

            Console.WriteLine(Enumerable.Empty<int>().Count());  // 0
            Console.WriteLine(Enumerable.Empty<int>() == null);  // false
            Console.WriteLine(Enumerable.Empty<int>().SequenceEqual(new int[]
{ }));  // true
```

```
            // Range - opretter et interval:

            foreach (int i in Enumerable.Range(3, 5))
                Console.WriteLine(i);  // 3, 4, 5, 6, 7
            Console.WriteLine(Enumerable
                .Range(5, 3).SequenceEqual(new int[] { 5, 6, 7 }));  // true

            // Repeat:

            foreach (string s in Enumerable.Repeat<string>("abc", 3))
                Console.WriteLine(s);  // abc, abc, abc
```

Efterord

Man kan komme ret langt med LINQ uden at kende til detaljerne nedenunder motorhjelmen. Hvis man vil være ekspert skal man dog nok i bøgerne eller onlinehjælpen.

Personligt vil jeg anbefale:

C# 3.0 in a Nutshell
O'Reilly
2007
Joseph Albahari & Ben Albahari
ISBN: 978-0-596-52757-0

Specielt kapitlerne 8 som dækker LINQ generelt og 9 som dækker LINQ operatorerne.

.oOo.

Pro LINQ
Language Integrated Query in C# 2008
Apress
Joseph C. Rattz, Jr.
ISBN: 978-1-59059-789-7

Der er næppe noget som er værd at vide om LINQ som ikke er dækket af denne bog!

Bogen indeholder masser af fornuftige kode eksempler. Han synes dog at have en forkærlighed for extension method syntax på bekostning af query comprehension syntax.

.oOo.

101 LINQ Samples:

http://msdn2.microsoft.com/en-us/vcsharp/aa336746.aspx

Eksemplerne bærer dog lidt præg af at de er skrevet før at LINQ blev endeligt frigivet med VS 2008.

Kommentar af zapzone d. 22. Mar 2008 | 1

Umiddelbart ganske god artikkel. Har dog ikke nærlæst den.

Interesserede sjæle skulle nu tage et kig på SubSonic. Uden tvivl noget af det fedeste jeg har oplevet inden for .NET... Super super nemt. Super super smart... :D :D

Tjek det ud her:
http://subsonicproject.com/

Jeg kan anbefalde deres web casts, for at få et hurtigt indblik i hvad det drejer sig om. Og det er altså ikke kun til Websites/web applications. Det egner sig lige så meget til winforms.

Kommentar af dj-hupi d. 07. May 2008 | 2

Fin artikel, meget gennem arbejdet!