.NET 3.5 og C# 3.0

Denne artikel viser hvordan man kan bruge nogle af de nye features i .NET 3.5 og C# 3.0. Det er ikke en komplet oversigt.

Den forudsætter et vist kendskab til .NET og C#.

Skrevet den **14. feb 2010** af **arne_v** I kategorien **Programmering / C#** | ★★★☆☆

Historie:
V1.0 - 22/11/2007 - original
V1.1 - 26/12/2008 - små ændringer
V1.2 - 14/02/2010 - smårettelser

Indledning

Jeg vil beskrive nogle af de nye features i .NET 3.5 og C# 3.0,
forklare hvad de kan bruges til og vise nogle meget simple eksempler.

Mine eksempler vil typisk være simplere end dem i dokumentationen.

Der er skrevet rigtigt meget om .NET 3.5 og C# 3.0, og hvis du har
læst det meste, så finder du næppe meget nyt i denne artikel.

Versioner

Lad os starte med at rede trådene med hensyn til versioner ud.

I 2005 udkom:
- C# 2.0 compiler
- .NET 2.0 med runtime og library

(se evt. artiklen http://www.eksperten.dk/guide/694 onkring NET 2.0 og C# 2.0)

I 2006 udkom:
- .NET 3.0 med ekstra libraries

(men bruger stadig 2.0 runtime og 2.0 libraries)

Nu (i 2007) er lige udkommet:
- C# 3.0 compiler
- .NET 3.5 med ekstra libraries

(men bruger stadig 2.0 runtime og 2.0 + 3.0 libraries)

TimeZoneInfo

En mangel i tidligere versioner af .NET har været den manglende time zone support.

.NET 3.5 giver mulighed for at slå en time zone op på navn.

Et lille eksempel.

tz.cs

```
using System;

public class TZ
{
    public static void Main(string[] args)
    {
        // finde time zone for computer (har man altid kunnet)
        TimeZone tz = TimeZone.CurrentTimeZone;
        Console.WriteLine(tz.StandardName + " " +
tz.GetUtcOffset(DateTime.Now));
        // ny måde at gøre det samme på i 3.5
        TimeZoneInfo tzi1 = TimeZoneInfo.Local;
        Console.WriteLine(tzi1.StandardName + " " +
tzi1.GetUtcOffset(DateTime.Now));
        // find vilkårlig time zone (nyt i 3.5)
        TimeZoneInfo tzi2 = TimeZoneInfo.FindSystemTimeZoneById("Eastern
Standard Time");
        Console.WriteLine(tzi2.StandardName + " " +
tzi2.GetUtcOffset(DateTime.Now));
    }
}
```

Nyttigt for visse typer applikationer.

HashSet

.NET 3.5 har fået en ny collection HashSet der kan bruges til set operationer. Eller på dans: mængde operationer.

Nedenstående eksempel viser de elementære operationer:

HS.cs

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

public class HS
{
    public static void Main(string[] args)
    {
        HashSet<int> hs1 = new HashSet<int>();
        hs1.Add(1);
        hs1.Add(2);
        hs1.Add(3);
        HashSet<int> hs2 = new HashSet<int>();
        hs2.Add(3);
        hs2.Add(4);
```

```
        foreach(int v in hs1) Console.Write(" " + v);
        Console.WriteLine();
        foreach(int v in hs2) Console.Write(" " + v);
        Console.WriteLine();
        // duplikater ignoreres
        hs1.Add(2);
        hs1.Add(3);
        foreach(int v in hs1) Console.Write(" " + v);
        Console.WriteLine();
        // forenings mængde
        HashSet<int> hs3 = new HashSet<int>(hs1.Union(hs2));
        foreach(int v in hs3) Console.Write(" " + v);
        Console.WriteLine();
        // fælles mængde
        HashSet<int> hs4 = new HashSet<int>(hs1.Intersect(hs2));
        foreach(int v in hs4) Console.Write(" " + v);
        Console.WriteLine();
        // komplementær mængde
        HashSet<int> hs5 = new HashSet<int>(hs1.Except(hs2));
        foreach(int v in hs5) Console.Write(" " + v);
        Console.WriteLine();
    }
}
```

Et særdeles nyttig funktionalitet i mange sammenhænge.

Pipes

.NET 3.5 har support for named pipes som et alternativ til TCP/IP sockets.

Lad os først se en client/server demo med traditionel TCP/IP sockets.

nserver.cs

```
using System;
using System.IO;
using System.Net;
using System.Net.Sockets;

public class NServer
{
    public static void Main(string[] args)
    {
        TcpListener srv = new TcpListener(IPAddress.Any, 1234);
        srv.Start();
        using(TcpClient cli = srv.AcceptTcpClient())
        {
            using(StreamReader sr = new StreamReader(cli.GetStream()))
            {
                for(int i = 0; i < 10000000; i++)
                {
```

```
                    if(sr.ReadLine() != "Dette er en test")
                    {
                        throw new Exception("Ooops");
                    }
                }
            }
        }
    }
}
```

nclient.cs

```
using System;
using System.IO;
using System.Net.Sockets;

public class NClient
{
    public static void Main(string[] args)
    {
        using(TcpClient cli = new TcpClient("localhost", 1234))
        {
            using(StreamWriter sw = new StreamWriter(cli.GetStream()))
            {
                for(int i = 0; i < 10000000; i++)
                {
                    sw.WriteLine("Dette er en test");
                }
            }
        }
    }
}
```

Så tager vi samme funktionalitet med named pipes.

pserver.cs

```
using System;
using System.IO;
using System.IO.Pipes;

public class PServer
{
    public static void Main(string[] args)
    {
        using(NamedPipeServerStream pipe = new NamedPipeServerStream("test"))
        {
            pipe.WaitForConnection();
```

```
            using(StreamReader sr = new StreamReader(pipe))
            {
                for(int i = 0; i < 10000000; i++)
                {
                    if(sr.ReadLine() != "Dette er en test")
                    {
                        throw new Exception("Ooops");
                    }
                }
            }
        }
    }
}
```

pclient.cs

```
using System;
using System.IO;
using System.IO.Pipes;

public class PClient
{
    public static void Main(string[] args)
    {
        using(NamedPipeClientStream pipe = new NamedPipeClientStream("test"))
        {
            pipe.Connect();
            using(StreamWriter sw = new StreamWriter(pipe))
            {
                for(int i = 0; i < 10000000; i++)
                {
                    sw.WriteLine("Dette er en test");
                }
            }
        }
    }
}
```

Hvorvidt det er bedest at bruge named pipes eller TCP/IP sockets er et meget komplekst spørgsmål. Der er forskellig funktionalitet, sikkerhed og performance karakteristikker.

Men arbejder man f.eks. med client/server løsninger med både client og server på samme system, så bør man bruge lidt tid på at se om named pipes er en god løsning for en.

Jeg tror på at denne feature får en vis udbredelse.

Automatic properties

C# 3.0 har en ny kortere syntax for standard properties.

Man plejer at lave fields og properties som følger.

p20.cs

```
public class P
{
    private int a;
    private string b;
    public int A
    {
        get
        {
            return a;
        }
        set
        {
            a = value;
        }
    }
    public string B
    {
        get
        {
            return b;
        }
        set
        {
            b = value;
        }
    }
}
```

Nu kan man skriv det samme meget kortere.

p30.cs

```
public class P
{
    public int A
    {
        get;
        set;
    }
    public string B
    {
        get;
        set;
    }
```

```
}
```

Bemærk at der genereres den samme kode for de 2 eksempler.

Jeg er 112% sikker på at den feature vil blive meget populær blandt programmørerne.

Extension methods

C# 3.0 har fået en nyttig lille feature til at tilføje metoder til andre klasser. Selve ideeen er vel nærmest AOP'sk, men implementeringen er meget nem.

Lad os lave et lille eksempel hvor vil tilføjer noget formatering:

ext.cs

```
using System;

public static class MyExtensions
{
    public static string ToHexString(this int o)
    {
        return o.ToString("X");
    }
    public static string ToVmsString(this DateTime o)
    {
        return o.ToString("dd-MMM-yyyy HH:mm");
    }
}

public class Test
{
    public static void Main(string[] args)
    {
        int i = 123;
        Console.WriteLine(i.ToHexString());
        DateTime dt = DateTime.Now;
        Console.WriteLine(dt.ToVmsString());
    }
}
```

En nyttig feature i mange situationer. Jeg vil dog fraråde at bruge den alt for meget, da det reelt vil lave en hel .NET library fork.

Shorthands

C# 3.0 har fået nogle features til mere kompakt kode til initialisering af både collections og objekter.

Las os først se noget traditionel 2.0 kode.

i20.cs

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

public class I20
{
    public static void Main(string[] args)
    {
        // objekt initialisering
        P o = new P();
        o.A = 123;
        o.B = "ABC";
        Console.WriteLine(o.A + " " + o.B);
        // collection initialisering
        List<int> lsti = new List<int>();
        lsti.Add(1);
        lsti.Add(2);
        lsti.Add(3);
        foreach(int v in lsti) Console.WriteLine(v);
        // kombineret
        List<P> lstp = new List<P>();
        P o1 = new P();
        o1.A = 123;
        o1.B = "ABC";
        lstp.Add(o1);
        P o2 = new P();
        o2.A = 456;
        o2.B = "DEF";
        lstp.Add(o2);
        foreach(P v in lstp) Console.WriteLine(v.A + " " + v.B);
    }
}
```

Og nu den samme kode i 3.0 version.

i30.cs

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

public class I30
{
    public static void Main(string[] args)
    {
        // objekt initialisering
        P o = new P{A = 123, B = "ABC" };
        Console.WriteLine(o.A + " " + o.B);
```

```
        // collection initialisering
        List<int> lsti = new List<int>{1, 2, 3};
        foreach(int v in lsti) Console.WriteLine(v);
        // kombineret
        List<P> lstp = new List<P>{new P{A = 123, B = "ABC" }, new P{A = 456,
B = "DEF" }};
        foreach(P v in lstp) Console.WriteLine(v.A + " " + v.B);
    }
}
```

To noter:
 - klassen P er den samme som ovenfor
 - 2.0 måden kan naturligvis forbedres lidt ved at have en constructor med argumenter

Men igen en feature som jeg er sikker på vil blive meget populær blandt programmørerne.

var type

C# 3.0 introducerer type interferens d.v.s. at typen af en venstre side variabel bestemmes udfra typen af højre side.

Simpelt eksempel.

v30.cs

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

public class V30
{
    public static void Main(string[] args)
    {
        var iv = 123;
        var sv = "ABC";
        Console.WriteLine(iv + " " + sv);
    }
}
```

Bemærk at var er ikke ekvivalent med object.

v20.cs

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

public class V20
{
```

```
    public static void Main(string[] args)
    {
        object iv = 123;
        object sv = "ABC";
        Console.WriteLine(iv + " " + sv);
    }
}
```

er en helt anden kode.

Forskellen ses ved at:

v20m.cs

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

public class V20M
{
    public static void Main(string[] args)
    {
        object iv = 123;
        object sv = "ABC";
        iv = "DEF";
        sv = 456;
        Console.WriteLine(iv + " " + sv);
    }
}
```

virker fint, mens:

v30m.cs

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

public class V30M
{
    public static void Main(string[] args)
    {
        var iv = 123;
        var sv = "ABC";
        iv = "DEF";
        sv = 456;
        Console.WriteLine(iv + " " + sv);
    }
}
```

giver:

v30m.cs(10,14): error CS0029: Cannot implicitly convert type 'string' to 'int'
v30m.cs(11,14): error CS0029: Cannot implicitly convert type 'int' to 'string'

var er med andre ord type safe !

I sig selv er var typen ikke specielt interessant. Ingen C# programmør vil vel
kode som ovenfor. Men var typen giver mening sammen med med LINQ (se nedenfor).

Lambda expressions

Lambda expressions er en meget omtalt ny feature i C# 3.0.

Den ligger meget godt i forlængelse af udviklingen i C#.

Las os starte med et C# 1.0 eksmepel.

solv1.cs

```
using System;

public class Solv
{
    // Newtons formel for iterativ lignings løsning
    private const double DELTA = 0.0000001;
    public delegate double FX(double x);
    public static double FindZero(FX f)
    {
        double x, xnext = 0;
        do
        {
            x = xnext;
            xnext = x - f(x) / ((f(x + DELTA) - f(x)) / DELTA);
        } while(Math.Abs(xnext - x) > DELTA);
        return xnext;
    }
    // simple trediegrads polynomium y=x^3+x-30 med en enkelt løsning x=3
    public static double MyPoly(double x)
    {
        return x * x * x + x - 30;
    }
    // løs
    public static void Main(string[] args)
    {
        Console.WriteLine(Solv.FindZero(MyPoly));
    }
}
```

Så går vi til C# 2.0 med anonymous methods.

solv2.cs

```
using System;

public class Solv
{
    // Newtons formel for iterativ lignings løsning
    private const double DELTA = 0.0000001;
    public delegate double FX(double x);
    public static double FindZero(FX f)
    {
        double x, xnext = 0;
        do
        {
            x = xnext;
            xnext = x - f(x) / ((f(x + DELTA) - f(x)) / DELTA);
        } while(Math.Abs(xnext - x) > DELTA);
        return xnext;
    }
    // løs simple trediegrads polynomium y=x^3+x-30 med en enkelt løsning x=3
    public static void Main(string[] args)
    {
        Console.WriteLine(Solv.FindZero(delegate(double x) { return x * x * x
+ x - 30; }));
    }
}
```

Og nu har vi så C# 3.0 med lambda expressions.

solv3.cs

```
using System;

public class Solv
{
    // Newtons formel for iterativ lignings løsning
    private const double DELTA = 0.0000001;
    public delegate double FX(double x);
    public static double FindZero(FX f)
    {
        double x, xnext = 0;
        do
        {
            x = xnext;
            xnext = x - f(x) / ((f(x + DELTA) - f(x)) / DELTA);
        } while(Math.Abs(xnext - x) > DELTA);
        return xnext;
    }
```

```
    // løs simple trediegrads polynomium y=x^3+x-30 med en enkelt løsning x=3
    public static void Main(string[] args)
    {
        Console.WriteLine(Solv.FindZero((double x) => x * x * x + x - 30 ));
    }
}
```

Syntaxen er blevet pænere. Så jeg er sikker på at programmørerne vil bruge denne feature. Men jeg synes ikke at det er nogen betydningsfuld ændring - ihvertfald ikke så betydningsfuld som foromtalen har antydet.

LINQ

Ingen ny feature i C# 3.0 har været så omtalt som LINQ.

LINQ = Language INtegrated Query

Sådan på jævnt dansk: brug af SQL lignende syntax i ens C# kode.

Det er et meget stort emne, så jeg kan kun komme ind på en meget lille del af alle mulighederne.

Men nok snik snak - nogle simple eksempler fortæller meget mere.

Jeg vil tillade mig at bruge diverse tidligere beskrevne features i eksemplerne.

Lad os starte med LINQ for Objects.

Vi genbruger igen vores P klasse.

linq1.cs

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

public class LINQforObject
{
    public static void Main(string[] args)
    {
        List<P> lstp = new List<P>{new P{A = 123, B = "ABC" }, new P{A = 456,
B = "DEF" }};
        // select en række (af typen P)
        var q1 = from o in lstp where o.A == 123 select o;
        foreach(var o in q1) Console.WriteLine(o.A + " " + o.B);
        // select en enkelt kolonne (af type string)
        var q2 = from o in lstp where o.A == 123 select o.B;
        foreach(var o in q2) Console.WriteLine(o);
        // select dele af en række (af anonym type)
        var q3 = from o in lstp where o.A == 123 select new { o.A, o.B};
```

```
        foreach(var o in q3) Console.WriteLine(o.A + " " + o.B);
        // select med aggregeret funktion til enkelt værdi (af typen decimal)
        var q4 = (from o in lstp select o.A).Average();
        Console.WriteLine(q4);
    }
}
```

Den kode kunne dog også laves lidt mere traditionelt.

linq2.cs

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

public class LINQforObject
{
    public static void Main(string[] args)
    {
        List<P> lstp = new List<P>{new P{A = 123, B = "ABC" }, new P{A = 456,
B = "DEF" }};
        // select en række (af typen P)
        var q1 = lstp.Where(o => o.A == 123).Select(o => o);
        foreach(var o in q1) Console.WriteLine(o.A + " " + o.B);
        // select en enkelt kolonne (af type string)
        var q2 = lstp.Where(o => o.A == 123).Select(o => o.B);
        foreach(var o in q2) Console.WriteLine(o);
        // select dele af en række (af anonym type)
        var q3 = lstp.Where(o => o.A == 123).Select( o => new { o.A, o.B});
        foreach(var o in q3) Console.WriteLine(o.A + " " + o.B);
        // select med aggregeret funktion til enkelt værdi (af typen decimal)
        var q4 = lstp.Select(o => o.A).Average();
        Console.WriteLine(q4);
    }
}
```

Vi skifter nu til LINQ i forbindelse med databaser.

Eksemplet vil bruge en test tabel som laves med:

```
GO
CREATE TABLE T1 (F1 INTEGER NOT NULL PRIMARY KEY, F2 VARCHAR(50))
GO
INSERT INTO T1 VALUES(1, 'A')
GO
INSERT INTO T1 VALUES(2, 'BB')
GO
INSERT INTO T1 VALUES(3, 'CCC')
GO
```

Vi kan nu lave den samme kode som før bare mod database.

linq3.cs

```
using System;
using System.Linq;
using System.Data.Linq;
using System.Data.Linq.SqlClient;
using System.Data.Linq.Mapping;
using System.Data.SqlClient;

public class LINQforDatabase
{
    [Table(Name="T1")]
    public class T1
    {
        [Column(IsPrimaryKey=true)]
        public int F1;
        [Column]
        public string F2;
    }
    public static void Main(string[] args)
    {
        SqlConnection con = new
SqlConnection(@"Server=ARNEPC3\SQLEXPRESS;Integrated
Security=SSPI;Database=Test");
        DataContext db = new DataContext(con);
        Table<T1> t1 = db.GetTable<T1>();
        // select en række (af typen T1)
        var q1 = from r in t1 where r.F1 == 2 select r;
        foreach(var r in q1) Console.WriteLine(r.F1 + " " + r.F2);
        // select en enkelt kolonne (af type string)
        var q2 = from r in t1 where r.F1 == 2 select r.F2;
        foreach(var r in q2) Console.WriteLine(r);
        // select dele af en række (af anonym type)
        var q3 = from r in t1 where r.F1 == 2 select new { r.F1, r.F2};
        foreach(var r in q3) Console.WriteLine(r.F1 + " " + r.F2);
        // select med aggregeret funktion til enkelt værdi (af typen double)
        var q4 = (from r in t1 select r.F1).Average();
        Console.WriteLine(q4);
        con.Close();
    }
}
```

Der er meget mere i dette store emne.

Videre læsning kan f.eks. starte her:
http://msdn2.microsoft.com/en-us/library/bb425822.aspx

LINQ er an revoloutionerende nyskabelse. Og der er da også skrevet utroligt meget om det. Det store spørgsmål er: hvor meget vil det blive brugt i praksis? Jeg er ikke overbevist om at anvendelsen vil stå mål med hverken nyskabelses niveauet eller omtale niveauet. Men tiden vil vise om jeg får ret.

Andet

Resten af de nye features må du selv slå op i docs.

Som appetitvækker nævner jeg lige:
- peer to peer support
- LINQ for XML

God kode lyst.

SP1

I 2008 kom .NET 3.5 SP1 som introducerede Entity Framework og LINQ for Entities, som vil erstatte LINQ for SQL.

Kommentar af conrad d. 05. dec 2007 | 1

super

Kommentar af mikkelk d. 23. nov 2007 | 2

Rigtig godt med en gennemgang af de nye ting i .NET 3.5, C# 3.0 - det har jeg manglet.

Kommentar af sherlock d. 05. dec 2007 | 3

En dejlig appetitvækker :)

Kommentar af googolplex d. 28. dec 2007 | 4

Kommentar af uggi16 d. 06. dec 2007 | 5

Må.. må.. må have mere af den slags :)

Kommentar af dustie d. 03. dec 2007 | 6

Kommentar af scheea2000 d. 26. nov 2007 | 7

Super! En hurtig, overskuelig og 'letfordøjelig' gennemgang.

Kommentar af kimr d. 02. dec 2007 | 8

Lækker gennemgang af de nye ting i .Net 3.5, C# 3.0
Det er godt med en nemt overskuelig artikel.

Kommentar af srofhest d. 28. nov 2007 | 9

Super lækker artikkel. Flot skrevet og nem at forstå selv for begyndere. Skal helt sikkert have fat i de nye

versioner.

Kommentar af henrykwdk d. 23. nov 2007 | 10

Det er meget fint - lige i øjet. Tak - det var også lige hvad jeg manglede. Meget appetitvækkende!