.NET 4.0 og C# 4.0

Denne artikel beskriver nogle af de nye features i .NET 4.0 og C# 4.0.

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

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

Historie:
V1.0 - 11/11/2009 - original
V1.1 - 12/02/2010 - smårettelser

Versions numre

Nu er der igen synkronisering af versions numrene.

C# går fra 3.0 til 4.0

.NET library går fra 3.5 til 4.0

.NET runtime går fra 2.0 til 4.0

Så alt er nu 4.0.

Hvornår

.NET 4.0 (og Visual Studio 2010) forventes releaset 12. april 2010.

Eksemplerne er testet med 4.0 beta 2. Potentielt kan der ske ændringer inden final.

Optional parametre med typ navn = def

C# har fået optional parametre ligesom C++ og forskellige andre sprog.

Man plejer at skulle bruge overload:

```
using System;

public class DefParm
{
    private int v;
    public DefParm() : this(0)
    {
    }
    public DefParm(int v)
    {
        this.v = v;
    }
```

```
    public override string ToString()
    {
        return "v:" + v;
    }
    public static void Main(string[] args)
    {
        Console.WriteLine(new DefParm());
        Console.WriteLine(new DefParm(1));
    }
}
```

Nu kan man angive en default værdi:

```
using System;

public class DefParm
{
    private int v;
    public DefParm(int v = 0)
    {
        this.v = v;
    }
    public override string ToString()
    {
        return "v:" + v;
    }
    public static void Main(string[] args)
    {
        Console.WriteLine(new DefParm());
        Console.WriteLine(new DefParm(1));
    }
}
```

Det gælder naturligvis alle metoder ikke kun constructor.

Praktisk lille feature.

Named paramtere med navn:val

For at kunne udnytte ovenstående fuldtud har man også givet mulighed for at angive parametere via navn i.s.f. via position.

Før:

```
using System;

public class NamParm
{
```

```
    private int v1;
    private int v2;
    private int v3;
    public NamParm() : this(0, 0, 0)
    {
    }
    public NamParm(int v1, int v2, int v3)
    {
        this.v1 = v1;
        this.v2 = v2;
        this.v3 = v3;
    }
    public override string ToString()
    {
        return "v1:" + v1 + " v2:" +v2 + " v3:" + v3;
    }
    public static void Main(string[] args)
    {
        Console.WriteLine(new NamParm());
        Console.WriteLine(new NamParm(0, 1, 0));
    }
}
```

Nu:

```
using System;

public class NamParm
{
    private int v1;
    private int v2;
    private int v3;
    public NamParm(int v1 = 0, int v2 = 0, int v3 = 0)
    {
        this.v1 = v1;
        this.v2 = v2;
        this.v3 = v3;
    }
    public override string ToString()
    {
        return "v1:" + v1 + " v2:" +v2 + " v3:" + v3;
    }
    public static void Main(string[] args)
    {
        Console.WriteLine(new NamParm());
        Console.WriteLine(new NamParm(v2:1));
    }
}
```

Praktisk lille feature.

dynamic & DLR

C# har fået en ny type dynamic som kan være hvad som helst.

Og hvordan adskiller den sig så fra object og var?

Lad os prøve at sammenligne dem.

Først noget helt normalt kode:

```
using System;

public class Std
{
    public static void Main(String[] args)
    {
        int o1 = 123;
        string o2 = "ABC";
        o1 = 123.456; // not allowed to ref to another type
        string s1x = o1.Substring(1); // not allowed to use methods not in
actual type
        string s2x = o2.Substring(1); // allowed to use methods in actual type
    }
}
```

Object kan man assigne hvad som helst til, men man kan kun bruge de metoder
der er i object.

```
using System;

public class Obj
{
    public static void Main(String[] args)
    {
        object o1 = 123;
        object o2 = "ABC";
        o1 = 123.456; // allowed to ref to another type
        string s1x = o1.Substring(1); // not allowed to use methods not in
object
        string s2x = o2.Substring(1); // not allowed to use methods not in
object
    }
}
```

C# 3.0 fil så den nye var type. Men var fungerer faktisk fuldstændigt ligesom
den helt normale kode.

Eneste forskel er at compileren selv bestemmer typen udfra højresiden i.s.f. at programmøren gør det.

```
using System;

public class Var
{
    public static void Main(String[] args)
    {
        var o1 = 123;
        var o2 = "ABC";
        o1 = 123.456; // not allowed to ref to another type
        string s1x = o1.Substring(1); // not allowed to use methods not in
actual type
        string s2x = o2.Substring(1); // allowed to use methods in actual type
    }
}
```

Og hvis nogen (som ikke er så kendte med C# 3.0) nu undrer sig over om C# programmører virkeligt er så dovne at det er et problem at skrive typen selv, så skal det lige understreges at den virkelige pointe i var er at den kan bruges til anonyme typer uden navn.

I dette eksempel kan programmøren ikke angive et navn:

using System;

```
public class WhyVar
{
    public static void Main(String[] args)
    {
        var z = new { A=123, B="ABC" };
        Console.WriteLine(z.A + " " + z.B);
    }
}
```

Og nu er vi så fremme ved C# 4.0 og dynamic.

Typen dynamic kan man assigne hvad som helst til. Og man kan tilgå hvilke som helst members (fields/properties/methods) på den - compileren checker ikke. Der sker først et check runtime.

```
using System;

public class Dyn
{
```

```
    public static void Main(String[] args)
    {
        dynamic o1 = 123;
        dynamic o2 = "ABC";
        o1 = 123.456; // allowed to ref to another type
        string s1x = o1.Substring(1); // allowed to use methods not in actual
type (but will generate runtime execption !)
        string s2x = o2.Substring(1); // allowed to use methods in actual type
    }
}
```

Jeg formoder ikke at dynamic bliver specielt populær i C#. Eneste område hvor det virkeligt vil være praktisk er i forbindelse med COM interop.

Men den infrastruktur der ligger bag ved i .NET - DLR (Dynamic Language Runtime) - er imidlertid særdeles nyttig når der skal implementeres dynamic typed sprog til .NET platformen.

Covariance og Contravariance

C# 4.0 tillader covariance og contravariance for generiske interfaces og delegates, hvis de erklærer at opfylde en kontrakt og faktisk overholder den kontrakt.

Det kan stort set kun forståes ved at kigge på et par eksempler. Eksemplerne er lidt lange, men det er en kompleks problem stilling.

Først covariance.

Her er noget C# 3.0 kode som ikke virker:

```
using System;

public class P
{
    public void M()
    {
        Console.WriteLine("M here");
    }
}

public class C : P
{
}

public interface IFoobar<T>
{
    T GetOne();
}

public class Foobar<T> : IFoobar<T> where T : new()
```

```
{
    public T GetOne()
    {
        return new T();
    }
}

public class Program
{
    public static void Main(String[] args)
    {
       IFoobar<P> fb = new Foobar<C>(); // compile error
       IFoobar<P> fb = (IFoobar<P>)new Foobar<C>(); // runtime error
       P o = fb.GetOne();
       o.M();
    }
}
```

Det er simpelthen ikke muligt at lave den asignment. Af meget gode grunde - med andet indhold af IFoobar/Foobar kunne man break type sikkerheden hvis det var muligt.

Men nu kommer C# 4.0:

```
using System;

public class P
{
    public void M()
    {
        Console.WriteLine("M here");
    }
}

public class C : P
{
}

public interface IFoobar<out T>
{
    T GetOne();
}

public class Foobar<T> : IFoobar<T> where T : new()
{
    public T GetOne()
    {
        return new T();
    }
}
```

```
public class Program
{
    public static void Main(String[] args)
    {
       IFoobar<P> fb = new Foobar<C>();
       P o = fb.GetOne();
       o.M();
    }
}
```

Dette virker fordi:
* interfacet med <out T> lover kun at have T som output aldrig som input til sine metoder
* interfacet faktisk holdet dette løfte
* Main ved dette og da output af type C kan assignes til P så accepterer den det

Tilsvarende/omvendt med contravariance.

C# 3.0:

```
using System;

public class P
{
    public void M()
    {
        Console.WriteLine("M here");
    }
}

public class C : P
{
}

public interface IFoobar<T> where T : P
{
    void CallM(T o);
}

public class Foobar<T> : IFoobar<T> where T : P
{
    public void CallM(T o)
    {
        o.M();
    }
}

public class Program
{
    public static void Main(String[] args)
    {
       IFoobar<C> fb = new Foobar<P>(); // compile error
```

```
        IFoobar<C> fb = (IFoobar<C>)new Foobar<P>(); // runtime error
        fb.CallM(new C());
    }
}
```

virker ikke.

C# 4.0:

```
using System;

public class P
{
    public void M()
    {
        Console.WriteLine("M here");
    }
}

public class C : P
{
}

public interface IFoobar<in T> where T : P
{
    void CallM(T o);
}

public class Foobar<T> : IFoobar<T> where T : P
{
    public void CallM(T o)
    {
        o.M();
    }
}

public class Program
{
    public static void Main(String[] args)
    {
        IFoobar<C> fb = new Foobar<P>();
        fb.CallM(new C());
    }
}
```

virker fordi:
* interfacet med <in T> lover kun at have T som input aldrig som output til sine metoder
* interfacet faktisk holdet dette løfte
* Main ved dette og da input af type C er en valid P så accepterer den det

Jeg tror ikke at denne feature vil blive brugt af ret mange. Det er for specielle problem stillinger og for komplekst.

Parallel extensions

.NET 4.0 kommer med et nyt parallel task library som gør det nemmere at lave multithreaded programmer.

Ideen er meget simpel. Programmøren fortæller at noget skal paralleliseres og .NET finder så selv ud af hvor mange tråde den vil bruge og laver alt det praktiske med at starte tråde og vente på at de afslutter.

Meget lig med de parallel programmings features som har været brugt i Fortran til vektor og matrix beregninger i ca. 20 år.

Først et simpelt eksempel med hvordan for og foreach løkker kan paralleliseres.

Traditionel C# kode:

```
using System;

public class MainClass
{
    public static void Main(string[] args)
    {
        int[] a = new int[1000];
        int[] b = new int[1000];
        for(int i = 0; i < a.Length; i++)
        {
            a[i] = i;
        }
        for(int i = 0; i < b.Length; i++)
        {
            b[i] = a[i] + 1;
        }
        foreach(int bv in b)
        {
            if(bv <= 0) throw new Exception("Ooops");
        }
        Console.WriteLine(b[77]);
    }
}
```

Med .NET 4.0 Parallel:

```
using System;
using System.Threading.Tasks;

public class MainClass
{
```

```
    public static void Main(string[] args)
    {
        int[] a = new int[1000];
        int[] b = new int[1000];
        Parallel.For(0, a.Length, (i) => { a[i] = i; } );
        Parallel.For(0, b.Length, (i) => { b[i] = a[i] + 1; } );
        Parallel.ForEach(b, (bv) => { if(bv <= 0) throw new
Exception("Ooops"); } );
        Console.WriteLine(b[77]);
    }
}
```

Så et eksempel med mere generel kode som skal udføres parallelt.

Traditionel:

```
using System;
using System.Threading;

public class MainClass
{
    public static void Main(string[] args)
    {
        Thread t1 = new Thread(Run1);
        Thread t2 = new Thread(Run2);
        Thread t3 = new Thread(Run3);
        t1.Start();
        t2.Start();
        t3.Start();
        t1.Join();
        t2.Join();
        t3.Join();
    }
    public static void Run1()
    {
        Console.WriteLine("Run1 in " + Thread.CurrentThread.ManagedThreadId);
    }
    public static void Run2()
    {
        Console.WriteLine("Run2 in " + Thread.CurrentThread.ManagedThreadId);
    }
    public static void Run3()
    {
        Console.WriteLine("Run3 in " + Thread.CurrentThread.ManagedThreadId);
    }
}
```

.NET 4.0:

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class MainClass
{
    public static void Main(string[] args)
    {
        // first way
        Parallel.Invoke(Run1, Run2, Run3);
        // second way
        Task t1 = new Task(Run1);
        Task t2 = new Task(Run2);
        Task t3 = new Task(Run3);
        t1.Start();
        t2.Start();
        t3.Start();
        Task.WaitAll(t1, t2, t3);
    }
    public static void Run1()
    {
        Console.WriteLine("Run1 in " + Thread.CurrentThread.ManagedThreadId);
    }
    public static void Run2()
    {
        Console.WriteLine("Run2 in " + Thread.CurrentThread.ManagedThreadId);
    }
    public static void Run3()
    {
        Console.WriteLine("Run3 in " + Thread.CurrentThread.ManagedThreadId);
    }
}
```

Og hvis nogen undrer sig over hvorfor nogen vil bruge Task fremfor Parallel.Invoke, så giver Task altså en række ekstra muligheder.

Parallel extensions er også tilføjet til LINQ, således at man kan udfører LINQ multithreaded.

Tradionel LINQ:

```
using System;
using System.Linq;
using System.Threading;

public class MainClass
{
    public static void Main(string[] args)
    {
        int[] a = new int[] { 1, 4, 9, 25, 36, 49, 64, 81, 100 };
```

```
        int maxa = a.Max();
        Console.WriteLine(maxa);
        // same but with debug to see how it is actually done
        int xmaxa = a.Max((v) => X(v));
        Console.WriteLine(xmaxa);
    }
    public static int X(int v)
    {
        Console.WriteLine("checking " + v + " in " +
Thread.CurrentThread.ManagedThreadId);
        return v;
    }
}
```

Parallel LINQ:

```
using System;
using System.Linq;
using System.Threading;

public class MainClass
{
    public static void Main(string[] args)
    {
        int[] a = new int[] { 1, 4, 9, 25, 36, 49, 64, 81, 100 };
        int maxa = a.AsParallel().Max();
        Console.WriteLine(maxa);
        // same but with debug to see how it is actually done
        int xmaxa = a.AsParallel().Max((v) => X(v));
        Console.WriteLine(xmaxa);
    }
    public static int X(int v)
    {
        Console.WriteLine("checking " + v + " in " +
Thread.CurrentThread.ManagedThreadId);
        return v;
    }
}
```

Jeg er sikker på at parallel extensions bliver en stor success. De gør det meget nemmere at lave (visse former for) multithreaded programmering og med 4/6/8 core CPU'er er multithreaded programmering et must.

POCO

.NET 3.5 SP1 tilføjede et nyt ORM til .NET kaldet EF (Entity Framework).

Men i første udgave skulle klasserne extende System.Data.Objects.DataClasses.EntityObject - med 4.0 kan man bruge EF med POCO's (Plain Old Clr Object's).

Jeg vil ikke gå i detaljer med dette, da EF er et stort område.

EF brugerne vil uden tvivl være glade for denne ændring.

Contracts

.NET 4.0 kommer med nogle nye og spændende features for at understøtte code contracts.

Lad os først se lidt kode:

```
using System;
using System.Diagnostics.Contracts;

public class Test
{
    private int v = 0;
    [ContractInvariantMethod]
    protected void ObjectInvariant()
    {
        Contract.Invariant(v < 100);
    }
    public void M1()
    {
        Contract.Requires(1 <= v && v <= 10);
        Console.WriteLine(v);
    }
    public void M2()
    {
        Contract.Ensures(1 <= v && v <= 10);
        Console.WriteLine(v);
    }
    public void M3()
    {
        Contract.EnsuresOnThrow<Exception>(1 <= v && v <= 10);
        throw new Exception("Ooops");
    }
    public void M4()
    {
        v = 200;
        Console.WriteLine(v);
    }
}

public class Program
{
    public static void Main(String[] args)
    {
        Test o = new Test();
        o.M1(); // violates pre condition
        o.M2(); // violates post condition
        o.M3(); // violates post condition
        o.M4(); // violates invariant
```

```
    }
}
```

Oplagt spørgsmål: hvordan adskiller dette sig fra traditionel if something throw new exception, forskellige assert kald etc.?

Svar: med alt! Dette er har meget lidt til fælles med de traditionelle tests.

De traditionelle tests:
- sættes ind alle de steder i koden hvor der skal testes
- testene compiles helt normalt
- testene enten udføres runtime eller er disablet
- fejl er helt normale exceptions

Contracts:
- conditions sættes ind et enkelt sted
- conditions compiles normalt men er ikke færdige efter normal kompilering
- der kan laves statisk analyse af korrektheden af den kompilede kode i forhold til conditions
- runtime check kan enables ved at modificere den allerede kompilerede kode (denne process sætter kode fra en enkelt condition ind alle de steder hvor der er brug for at teste)
- fejl er ikke normal exceptions og kan ikke catches (medmindre man erstatter en standard klasse med en custom klasse)

Både statisk analyse og build med runtime check kan nemt laves via Visual Studio.

Contracts er reelt en form for AOP (Aspect Oriented Programming).

Jeg tror ikke at denne her feature har fundet sin endelige udformning, men det ser meget interessant ud. Men desværre tvivler jeg lidt på at den vil blive brugt ret meget. Det er nok for komplekst.

Andet

Der er mange andre nye features, men dem må du selv slå op i docs.

God kode lyst.

Foregående artikler

http://www.eksperten.dk/guide/694 om .NET 2.0 og C# 2.0
http://www.eksperten.dk/guide/1153 on .NET 3.5 og C# 3.0

Kommentar af j3ppah d. 02. dec 2009 | 1

Bliver for fedt :D.
Fed artikkel og thumbs up, herfra! Keep up the good work!