Denne guide er oprindeligt
udgivet på Eksperten.dk

COMPUTERWORLD

Singleton pattern i C#

Denne artikel beskriver Singleton pattern og implementation i C#.

Den forudsætter kendskab til C# men ikke til Singleton. Der er en anden artikel med præcis samme indhold bare i VB.NET !

Skrevet den **09. Feb 2009** af **arne_v** I kategorien **Programmering / C#** | ★★★☆☆

Historie:
V1.0 - 12/01/2004 - original
V1.1 - 31/01/2004 - forbedret formatering
V1.2 - 17/02/2004 - tilføj henvisning til MSDN artikel om lock typeof
V1.3 - 07/03/2004 - tilføj henvisning til VB.NET artikel
V1.4 - 10/05/2005 - ændre til at double locking heller ikke er sikker i .NET

Teori

Singleton pattern løser problemet med at man kun vil have en enkelt instans af en given klasse.

Singleton pattern er en god objekt orienteret løsning på samme problem som løses ikke objekt orienteret via en klasse med kun static members og methods.

Singleton pattern er et såkaldt GoF pattern, hvilket refererer til bogen "Design Patterns" af Erich Gamma, Richard Helm, Ralph Johnson og John Vlissides (4 forfattere = Gang Of Four = GoF).

Kendetegnene ved en singleton klasse er:
- public static metode GetInstance eller property Instance
- private constructor

(den original GoF kode bruger protected constructor og det kan man også godt, men min erfaring er at man ikke kan arve fra en singleton klasse på fornuftig vis)

Eksempel

Her er et standard eksempel på en singleton klasse:

```
using System;
using System.Collections;

// singleton klasse
public class S1
{
    // normale attributter eksemplificeret ved en ArrayList
    private ArrayList list;
    // den eneste instans der eksisterer
    private static S1 instance = null;
    // private constructor
    private S1() {
```

```
        list = new ArrayList();
    }
    // public static metode til at hente instance
    public static S1 GetInstance()
    {
        if(instance == null)
        {
            instance = new S1();
        }
        return instance;
    }
    // normale metoder
    public void Add(object o)
    {
        list.Add(o);
    }
    public ArrayList GetList()
    {
        return list;
    }
}
```

eller:

```
using System;
using System.Collections;

// singleton klasse
public class S2
{
    // normale attributter eksemplificeret ved en ArrayList
    private ArrayList list;
    // den eneste instans der eksisterer
    private static S2 instance = new S2();
    // private constructor
    private S2() {
        list = new ArrayList();
    }
    // public static metode til at hente instance
    public static S2 GetInstance()
    {
        return instance;
    }
    // normale metoder
    public void Add(object o)
    {
        list.Add(o);
    }
    public ArrayList GetList()
    {
        return list;
    }
}
```

C# tillader imidlertid et elegant alternativ med brug af property:

```
using System;
using System.Collections;

public class S3
{
    // normale attributter eksemplificeret ved en ArrayList
    private ArrayList list;
    // den eneste instans der eksisterer
    private static S3 instance = null;
    // private constructor
    private S3() {
        list = new ArrayList();
    }
    // property til at hente instance
    public static S3 Instance
    {
        get {
            if(instance == null)
            {
                instance = new S3();
            }
            return instance;
        }
    }
    // normale metoder
    public void Add(object o)
    {
        list.Add(o);
    }
    public ArrayList GetList()
    {
        return list;
    }
}
```

Klassen kan bruges som følger:

```
using System;
using System.Collections;

class MainClass
{
    public static void Main(string[] args)
    {
        S3 a = S3.Instance;
        a.Add("A");
        S3 b = S3.Instance;
```

```
        b.Add("B");
        S3 c = S3.Instance;
        ArrayList list = c.GetList();
        for(int i = 0; i < list.Count; i++)
        {
            Console.WriteLine((string)list[i]);
        }
    }
}
```

Singleton i multithreaded kontekst
Bemærk at i en multithreaded kontekst bør man kode sin singleton klasse som:

```
using System;
using System.Collections;

// singleton klasse
public class S4
{
    // normale attributter eksemplificeret ved en ArrayList
    private ArrayList list;
    // den eneste instans der eksisterer
    private static S4 instance = null;
    // private constructor
    private S4() {
        list = new ArrayList();
    }
    // lock object
    private static object mylock = new object();
    // property til at hente instance
    public static S4 Instance
    {
        get {
            lock(mylock)
            {
                if(instance == null)
                {
                    instance = new S4();
                }
            }
            return instance;
        }
    }
    // normale metoder
    public void Add(object o)
    {
        list.Add(o);
    }
    public ArrayList GetList()
    {
        return list;
```

```
    }
}
```

Vær forsigtig med at bruge:

```
using System;
using System.Collections;

// singleton klasse
public class S5
{
    // normale attributter eksemplificeret ved en ArrayList
    private ArrayList list;
    // den eneste instans der eksisterer
    private static S5 instance = null;
    // private constructor
    private S5() {
        list = new ArrayList();
    }
    // lock object
    private static object mylock = new object();
    // property til at hente instance
    public static S5 Instance
    {
        get {
            if(instance == null)
            {
                lock(mylock)
                {
                    if(instance == null)
                    {
                        instance = new S5();
                    }
                }
            }
            return instance;
        }
    }
    // normale metoder
    public void Add(object o)
    {
        list.Add(o);
    }
    public ArrayList GetList()
    {
        return list;
    }
}
```

For selvom det er blevet hævdet af mange inkl. diverse

Microsoft eksempler, at dette trick kendt som double locing altid virker i .NET, så er fakta at det kun altid virker på IA-32 og AMD-64 kompatible CPU'er. Problemet kan løses ved enten at erklære instance field som volatile eller bruge Thread.MemoryBarrier, men så er en banal lock nok pænere. Der er dog ikke mange som kører .NET på IA-64 CPU'er endnu.

I .NET verdenen er det ildeset med brug af lock(typeof(Sx)) og det bør derfor undgåes (selvom argumentet måske er lidt tyndt) !

For detaljer om dette læs:
http://blogs.msdn.com/brada/archive/2004/05/12/130935.aspx
http://blogs.gotdotnet.com/cbrumme/PermaLink.aspx/480d3a6d-1aa8-4694-96db-c69f01d7ff2b
http://discuss.develop.com/archives/wa.exe?A2=ind0203B&L=DOTNET&P=R375
og:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaskdr/html/askgui06032003.asp

Kommentar af iakob d. 17. Jun 2004 | 1

Der er et godt, brugbart og simpelt eksempel på en implementation. Jeg synes artiklen godt kunne have undværet de "uelegante" løsninger (forfatteren har selv en favorit og jeg er enig i hans valg. Der er en enkelt påstand omkring noget man bør undgå, som der ikke er argumenteret for

Kommentar af repsak d. 16. Feb 2004 | 2

Mange eksempler :-) Ser dog ikke grund til at lave så mange næsten ens, selvom der er en god pointe/linje imellem de forskellige versioner og forbedringer.
Efter min mening mangler dog et argument for den sidste linje :-)

Kommentar af thundergod d. 07. May 2004 | 3

Artiklen lever op til overskriften.
Det er kun anvendelsen af Property, og lock(...) der er ny information.

Kommentar af jimgordon d. 04. Dec 2004 | 4

Perfekt miniartikel. Giver singleton fidusen i fin telegramstil, så man er klar til den tungere litterature omkring dette uundværlige programmeringsdesign.

Kommentar af innercitydk d. 27. Jul 2006 | 5

Super artikel. Så er der styr på singleton til bunds ;)

Kommentar af impero d. 29. Jun 2011 | 6

Ved godt dette er en gammel guide, men ville lige give mit input til nye læsere.

Den måde jeg foretrækker at lave singletons i C# er følgende:

```
class Singleton
{
   private Singleton()
   {
```

```
    }

    public static Singleton Instance
    {
        get
        {
            return Nested.Singleton;
        }
    }

    private class Nested
    {
        static Nested() { }
        internal static readonly Singleton Singleton =
            new Singleton();
    }
}
```

Denne metode er både thread-safe og lazy, hvilket betyder at den virker i multi-threaded programmer samt at den først opretter singleton objektet når det bliver refereret første gang. Derudover er den super simpel at implementere og kræver ikke brug af locks.

Kommentar af arne_v d. 31. Oct 2011 | 7

re impero)

Den teknik er ret kendt.

Men har du nogen sinde set et eksempel hvor overhead ved den lock er stort nok til at begrunde de ekstra linier kode??