Singleton pattern i Java

Denne artikel beskriver Singleton pattern og implementation i Java.

Den forudsætter kendskab til Java men ikke til Singleton.

Skrevet den **06. Mar 2009** af **arne_v** I kategorien **Programmering / Java** | ★★★☆☆

Historie:
V1.0 - 12/01/2004 - original
V1.1 - 31/01/2004 - forbedret formatering + lidt forklaringer om brug
V1.2 - 17/02/2004 - tilføj henvisning til IBM artikel om double locking

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
- 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)

Singleton bruges typisk til at indeholde en data struktur som der kun må eksistere en af p.g.a. den information den indeholder (pools, queues og den slags). Og har den fordel at man når som helst kan hente en referance til den uden at skulle sende den med over til constructor og gemme den der.

Eksempel

Her er et standard eksempel på en singleton klasse:

```
import java.util.*;

// singleton klasse
public class S1 {
    // normale attributter eksemplificeret ved en List
    private List 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 List getList() {
        return list;
    }
}
```

Alternativt kan den kodes som:

```
import java.util.*;

// singleton klasse
public class S2 {
    // normale attributter eksemplificeret ved en List
    private List 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 List getList() {
        return list;
    }
}
```

Den første form er dog mest udbredt, fordi den ligner den original GoF i C++ mere, og fordi det ikke er altid at man kan initialisere instansen på classload tidspunktet.

Klassen kan bruges som følger:

```
public class TestS {
    public static void main(String[] args) {
        S1 a = S1.getInstance();
        a.add("A");
        S1 b = S1.getInstance();
        b.add("B");
        S1 c = S1.getInstance();
        System.out.println(c.getList());
    }
}
```

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

```
import java.util.*;

// singleton klasse
public class S3 {
    // normale attributter eksemplificeret ved en List
    private List list;
    // den eneste instans der eksisterer
    private static S3 instance = null;
    // private constructor
    private S3() {
        list = new ArrayList();
    }
    // public static metode til at hente instance
    public static synchronized S3 getInstance() {
        if(instance == null) {
            instance = new S3();
        }
        return instance;
    }
    // normale metoder
    public synchronized void add(Object o) {
        list.add(o);
    }
    public List getList() {
        return list;
    }
}
```

En meget vigtig note er at varianten med såkaldt double locking:

```
import java.util.*;

// singleton klasse
```

```
public class S4 {
    // normale attributter eksemplificeret ved en List
    private List list;
    // den eneste instans der eksisterer
    private static S4 instance = null;
    // private constructor
    private S4() {
        list = new ArrayList();
    }
    // public static metode til at hente instance
    public static S4 getInstance() {
        if(instance == null) {
            synchronized(S4.class) {
                if(instance == null) {
                    instance = new S4();
                }
            }
        }
        return instance;
    }
    // normale metoder
    public synchronized void add(Object o) {
        list.add(o);
    }
    public List getList() {
        return list;
    }
}
```

på trods af at den umiddelbart ser smart ud ikke er thread safe i Java på trods af at den tidligere har været anbefalet forskellige steder.

For detaljer om dette læs:
http://www-106.ibm.com/developerworks/java/library/j-dcl.html?dwzone=java?e

Andre ting man skal være opmærksom på
En vigtig note er, at en singleton er unik indenfor en JVM d.v.s. at der kan sagtens forekomme flere instanser af en singleton klasse i forskellige processer (som køre i forskellige JVM's).

Og p.g.a. den måde Java identificerer en klasse på, så kan der faktisk godt være flere instanser af en singleton klasse indenfor samme JVM, hvis de er loadet af forskellige classloaders. Bemærk at dette er ikke en typisk sitaution. Langt de fleste J2SE applikationer vil kun bruge en enkelt classloader. Men i J2EE verdenen bruger man hyppigt flere classloaders. Hvis du påtænker at bruge singleton i J2EE environment, så læs dokumentation om din applikations servers brug af classloaders og overvej om den er kompatibel med dine intentioner for din singelton klasse.

Kommentar af simonvalter d. 16. Sep 2005 | 1

God artikel.
double locking er thread safe fra Java SE 5 og så kan det vist nævnes at 1. initialiseres senere, 2. med det samme og er iøvrigt også thread safe. 3. kan være langsommere end 4. da der kun synkroniseres ved oprettelse af instans... skal 4. iøvrigt ikke bruge volatile på instansen?

Kommentar af mercur8 (nedlagt brugerprofil) d. 21. Jan 2004 | 2

Fin og præcis artikel, flere gode detaljer er med. Mangler måske en kommentar om/beskrivelse af en singleton uden private constructor, så nedarvning understøttes (protected, exception kast i constructor). + Eksempler på brug (runtime, toolkit, db forbindelse etc.).

Kommentar af codemon d. 01. Feb 2004 | 3

Godt Arne, endnu en artikel som er nem at læse, godt forklaret. Ikke alt mulig udenomssnak. Bortset fra thread-safe singeton ses ofte public final static INSTANCE... så get metoden bliver overflødig.

Kommentar af digitalsoul d. 06. May 2004 | 4