Implementing the Singleton Pattern in C#

Introduction

The singleton pattern is one of the best-known patterns in software engineering. Essentially, a singleton is a class which only allows a single instance of itself to be created, and usually gives simple access to that instance. Most commonly, singletons don't allow any parameters to be specified when creating the instance - as otherwise a second request for an instance but with a different parameter could be problematic! (If the same instance should be accessed for all requests with the same parameter, the factory pattern is more appropriate.) This article deals only with the situation where no parameters are required. Typically a requirement of singletons is that they are created lazily - i.e. that the instance isn't created until it is first needed.

There are various different ways of implementing the singleton pattern in C#. I shall present them here in reverse order of elegance, starting with the most commonly seen, which is not thread-safe, and working up to a fully lazily-loaded, thread-safe, simple and highly performant version.

All these implementations share four common characteristics, however:

- A single constructor, which is private and parameterless. This prevents other classes from instantiating it (which would be a violation of the pattern). Note that it also prevents subclassing - if a singleton can be subclassed once, it can be subclassed twice, and if each of those subclasses can create an instance, the pattern is violated. The factory pattern can be used if you need a single instance of a base type, but the exact type isn't known until runtime.
- The class is sealed. This is unnecessary, strictly speaking, due to the above point, but may help the JIT to optimise things more.
- A static variable which holds a reference to the single created instance, if any.
- A public static means of getting the reference to the single created instance, creating one if necessary.

Note that all of these implementations also use a public static property `Instance` as the means of accessing the instance. In all cases, the property could easily be converted to a method, with no impact on thread-safety or performance.

First version - not thread-safe

```
// Bad code! Do not use!
public sealed class Singleton
{
    private static Singleton instance=null;

    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            if (instance==null)
            {
                instance = new Singleton();
            }
            return instance;
        }
    }
}
```

As hinted at before, the above is not thread-safe. Two different threads could both have evaluated the test `if (instance==null)` and found it to be true, then both create instances, which violates the singleton pattern. Note that in fact the instance may already have been created before the expression is evaluated, but the memory model doesn't guarantee that the new value of instance will be seen by other threads unless suitable memory barriers have been passed.

Second version - simple thread-safety

```
public sealed class Singleton
{
    private static Singleton instance = null;
    private static readonly object padlock = new object();

    Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
}
```

This implementation is thread-safe. The thread takes out a lock on a shared object, and then checks whether or not the instance has been created before creating the instance. This takes care of the memory barrier issue (as locking makes sure that all reads occur logically after the lock acquire, and unlocking makes sure that all writes occur logically before the lock release) and ensures that only one thread will create an instance (as only one thread can be in that part of the code at a time - by the time the second thread enters it,the first thread will have created the instance, so the expression will evaluate to false). Unfortunately, performance suffers as a lock is acquired every time the instance is requested.

Note that instead of locking on `typeof(Singleton)` as some versions of this implementation do, I lock on the value of a static variable which is private to the class. Locking on objects which other classes can access and lock on (such as the type) risks performance issues and even deadlocks. This is a general style preference of mine - wherever possible, only lock on objects specifically created for the purpose of locking, or which document that they are to be locked on for specific purposes (e.g. for waiting/pulsing a queue). Usually such objects should be private to the class they are used in. This helps to make writing thread-safe applications significantly easier.

Third version - attempted thread-safety using double-check locking

```
// Bad code! Do not use!
public sealed class Singleton
{
    private static Singleton instance = null;
    private static readonly object padlock = new object();

    Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            if (instance == null)
            {
                lock (padlock)
                {
                    if (instance == null)
                    {
                        instance = new Singleton();
                    }
                }
            }
            return instance;
        }
    }
}
```

This implementation attempts to be thread-safe without the necessity of taking out a lock every time. Unfortunately, there are four downsides to the pattern:

- It doesn't work in Java. This may seem an odd thing to comment on, but it's worth knowing if you ever need the singleton pattern in Java, and C# programmers may well also be Java programmers. The Java memory model doesn't ensure that the constructor completes before the reference to the new object is assigned to instance. The Java memory model underwent a reworking for version 1.5, but double-check locking is still broken after this without a volatile variable (as in C#).
- Without any memory barriers, it's broken in the ECMA CLI specification too. It's possible that under the .NET 2.0 memory model (which is stronger than the ECMA spec) it's safe, but I'd rather not rely on those stronger semantics, especially if there's any doubt as to the safety. Making the `instance` variable volatile can make it work, as would explicit memory barrier calls, although in the latter case even experts can't agree exactly which barriers are required. I tend to try to avoid situations where experts don't agree what's right and what's wrong!

- It's easy to get wrong. The pattern needs to be pretty much exactly as above - any significant changes are likely to impact either performance or correctness.
- It still doesn't perform as well as the later implementations.

Fourth version - not quite as lazy, but thread-safe without using locks

```
public sealed class Singleton
{
    private static readonly Singleton instance = new Singleton();

    // Explicit static constructor to tell C# compiler
    // not to mark type as beforefieldinit
    static Singleton()
    {
    }

    private Singleton()
    {
    }

    public static Singleton Instance
    {
        get
        {
            return instance;
        }
    }
}
```

As you can see, this is really is extremely simple - but why is it thread-safe and how lazy is it? Well, static constructors in C# are specified to execute only when an instance of the class is created or a static member is referenced, and to execute only once per AppDomain. Given that this check for the type being newly constructed needs to be executed whatever else happens, it will be faster than adding extra checking as in the previous examples. There are a couple of wrinkles, however:

- It's not as lazy as the other implementations. In particular, if you have static members other than `Instance`, the first reference to those members will involve creating the instance. This is corrected in the next implementation.
- There are complications if one static constructor invokes another which invokes the first again. Look in the .NET specifications (currently section 9.5.3 of partition II) for more details about the exact nature of type initializers - they're unlikely to bite you, but it's worth being aware of the consequences of static constructors which refer to each other in a cycle.
- The laziness of type initializers is only guaranteed by .NET when the type isn't marked with a special flag called `beforefieldinit`. Unfortunately, the C# compiler (as provided in the .NET 1.1 runtime, at least) marks all types which don't have a static constructor (i.e. a block which looks like a constructor but is marked static) as `beforefieldinit`. I now have an article with more details about this issue. Also note that it affects performance, as discussed near the bottom of the page.

One shortcut you can take with this implementation (and only this one) is to just make `instance` a public static readonly variable, and get rid of the property entirely. This makes the basic skeleton code absolutely tiny! Many people, however, prefer to have a property in case further action is needed in future, and JIT inlining is likely to make the performance identical. (Note that the static constructor itself is still required if you require laziness.)

Fifth version - fully lazy instantiation

```
public sealed class Singleton
{
    private Singleton()
    {
    }

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

    private class Nested
    {
        // Explicit static constructor to tell C# compiler
        // not to mark type as beforefieldinit
        static Nested()
        {
        }

        internal static readonly Singleton instance = new Singleton();
    }
}
```

Here, instantiation is triggered by the first reference to the static member of the nested class, which only occurs in `Instance`. This means the implementation is fully lazy, but has all the performance benefits of the previous ones. Note that although nested classes have access to the enclosing class's private members, the reverse is not true, hence the need for `instance` to be internal here. That doesn't raise any other problems, though, as the class itself is private. The code is a bit more complicated in order to make the instantiation lazy, however.

Sixth version - using .NET 4's `Lazy<T>` type

If you're using .NET 4 (or higher), you can use the System.Lazy<T> type to make the laziness really simple. All you need to do is pass a delegate to the constructor which calls the Singleton constructor - which is done most easily with a lambda expression.

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> lazy =
        new Lazy<Singleton>(() => new Singleton());

    public static Singleton Instance { get { return lazy.Value; } }

    private Singleton()
    {
    }
}
```

It's simple and performs well. It also allows you to check whether or not the instance has been created yet with the IsValueCreated property, if you need that.

Performance vs laziness

In many cases, you won't actually require full laziness - unless your class initialization does something particularly time-consuming, or has some side-effect elsewhere, it's probably fine to leave out the explicit static constructor shown above. This can increase performance as it allows the JIT compiler to make a single check (for instance at the start of a method) to ensure that the type has been initialized, and then assume it from then on. If your singleton instance is referenced within a relatively tight loop, this can make a (relatively) significant performance difference. You should decide whether or not fully lazy instantiation is required, and document this decision appropriately within the class.

A lot of the reason for this page's existence is people trying to be clever, and thus coming up with the double-checked locking algorithm. There is an attitude of locking being expensive which is common and misguided. I've written a very quick benchmark which just acquires singleton instances in a loop a billion ways, trying different variants. It's not terribly scientific, because in real life you may want to know how fast it is if each iteration actually involved a call into a method fetching the singleton, etc. However, it does show an important point. On my laptop, the slowest solution (by a factor of about 5) is the locking one (solution 2). Is that important? Probably not, when you bear in mind that it still managed to acquire the singleton a *billion* times in under 40 seconds. (Note: this article was originally written quite a while ago now - I'd expect better performance now.) That means that if you're "only" acquiring the singleton four hundred thousand times per second, the cost of the acquisition is going to be 1% of the performance - so improving it isn't going to do a lot. Now, if you *are* acquiring the singleton that often - isn't it likely you're using it within a loop? If you care that much about improving the

performance a little bit, why not declare a local variable outside the loop, acquire the singleton once and *then* loop. Bingo, even the slowest implementation becomes easily adequate.

I would be very interested to see a *real world* application where the difference between using simple locking and using one of the faster solutions actually made a significant performance difference.

Exceptions

Sometimes, you need to do work in a singleton constructor which may throw an exception, but might not be fatal to the whole application. Potentially, your application may be able to fix the problem and want to try again. Using type initializers to construct the singleton becomes problematic at this stage. Different runtimes handle this case differently, but I don't know of any which do the desired thing (running the type initializer again), and even if one did, your code would be broken on other runtimes. To avoid these problems, I'd suggest using the second pattern listed on the page - just use a simple lock, and go through the check each time, building the instance in the method/property if it hasn't already been successfully built.

Thanks to Andriy Tereshchenko for raising this issue.

Conclusion (modified slightly on January 7th 2006; updated Feb 12th 2011)

There are various different ways of implementing the singleton pattern in C#. A reader has written to me detailing a way he has encapsulated the synchronization aspect, which while I acknowledge may be useful in a few *very* particular situations (specifically where you want very high performance, *and* the ability to determine whether or not the singleton has been created, *and* full laziness regardless of other static members being called). I don't personally see that situation coming up often enough to merit going further with on this page, but please mail me if you're in that situation.

My personal preference is for solution 4: the only time I would normally go away from it is if I needed to be able to call other static methods without triggering initialization, or if I needed to know whether or not the singleton has already been instantiated. I don't remember the last time I was in that situation, assuming I even have. In that case, I'd probably go for solution 2, which is still nice and easy to get right.

Solution 5 is elegant, but trickier than 2 or 4, and as I said above, the benefits it provides seem to only be rarely useful. Solution 6 is a simpler way to achieve laziness, if you're using .NET 4. It also has the advantage that it's *obviously* lazy. I currently tend to still use solution 4, simply through habit - but if I were working with inexperienced developers I'd quite possibly go for solution 6 to start with as an easy and universally applicable pattern.

(I wouldn't use solution 1 because it's broken, and I wouldn't use solution 3 because it has no benefits over 5.)