Article

Singleton Design Pattern In C# - Part Two (Eager and Lazy Initialization in Singleton)

By **Akhil Mittal** on **Updated date Jan 11, 2018**

Introduction

In the previous article on learning singleton pattern, we discussed what is singleton design pattern, when is it needed and what problemsit solves. We also discussed how to create a basic singleton class step by step and how to enhance that class to be thread-safe with the help of locking and performance effective with the help of double check locking. In this article, we'll discuss Lazy initialization, the lazy keyword, why to make singleton class a sealed class, and what are the differences between singleton and static class. Before we start, I strongly recommend you go through my last article. Following is the three article series we are following to learn about singleton and static classes.

1. **Singleton Design Pattern In C# - Part One**
2. **Singleton Design Pattern In C# - Part Two**
3. **Singleton Design Pattern In C# - Part Three (Singleton vs Static)**

Lazy Initialization

In the last article, we discussed what is lazy initialization. Lazy initialization basically is very performance effective. In situations like on-demand object creation where we create an object of the class only when it is accessed, lazy initialization works very well. It helps application load faster because it does not depend upon instance creation at the time of application startup. There could also be situations where one would go for eager initialization. Eager initialization is opposite of lazy initialization or we can say non-lazy initialization. Let's see how we can make our singleton class as supporting eager initialization. You can get the source code we developed in our last article and get started. The same is attached along with the article as well.

Eager Initialization Implementation

Eager loading is basically initializing the object that is to be accessed in future and keeping it in memory rather than initializing it on demand so that we can use the object whenever we need those.

Open Visual Studio, the last solution of Singleton and open the Singleton class.

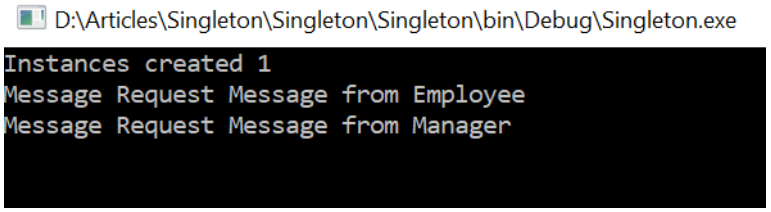
1. Change null backing field initialization to new Singleton class initialization as following and make the backing field read-only.

```
private static readonly Singleton singleInstance = new Singleton(); //private static Singleton singleInstance = null;
```

2. Since we made the backing field read-only, we cannot instantiate that again in our SingleInstance property, so just return the backing field as it is and remove the double check locking we implemented in our last version of the singleton.

```
2 references
public static Singleton SingleInstance
{
  get
  {
    return singleInstance;
  }
}
```

3. Remove the locking variable as well that we declared earlier. Now run the application and check the output. Note that we still have two methods invoked parallelly.



We see that only one instance got created. This is not surprising that our class is still thread-safe after removing the lock variables because the CLR internally takes care of variable initialization and thus save us from getting into deadlock situation in eager loading initialization.

Following is the complete code.

```
using static System.Console;
namespace Singleton {
    class class Singleton {
        static int instanceCounter = 0;
        private static readonly Singleton singleInstance = new Singleton(); //private static Singleton singleInstance = null;
        private Singleton() {
            instanceCounter++;
            WriteLine("Instances created " + instanceCounter);
        }
        public static Singleton SingleInstance {
            get {
                return singleInstance;
            }
        }
        public void LogMessage(string message) {
            WriteLine("Message " + message);
```

```
17.       }
18.    }
19. }
```

Lazy Initialization Implementation using Lazy Keyword

We can modify our class to lazily initialize using a lazy keyword. Let's check how we can do it.

1. First, change the backing field initialization to Lazy initialization be following code,

```
static int instanceCounter = 0;
private static readonly Lazy<Singleton> singleInstance = new Lazy<Singleton>(()=>new Singleton())
```

private static readonly Lazy<Singleton> singleInstance = new Lazy<Singleton>(()=>new Singleton());

This is the way we lazily initialize an object by passing the delegate to create instance as () => new Singleton()

2. Now, in the property, the instance could not be returned directly, but we return singleInstance.Value because now singleInstance.Value is Singleton class type and not the instance.

```
public static Singleton SingleInstance
{
  get
  {
    return singleInstance.Value;
  }
}
```

Full class code is as following.

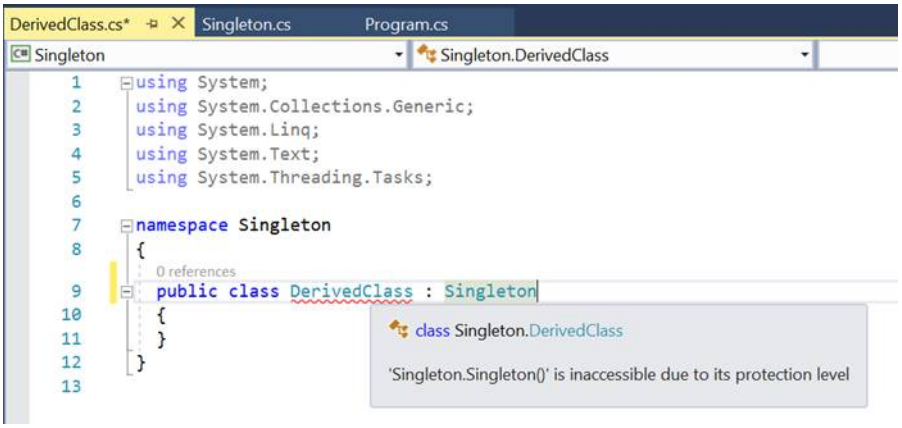
```
1. using System;
2. using static System.Console;
3. namespace Singleton {
4.     sealed class Singleton {
5.         static int instanceCounter = 0;
6.         private static readonly Lazy < Singleton > singleInstance = new Lazy < Singleton > (() => new Singleton()); //private static Singleton singleIns
7.         private Singleton() {
8.             instanceCounter++;
9.             WriteLine("Instances created " + instanceCounter);
10.        }
11.        public static Singleton SingleInstance {
12.            get {
13.                return singleInstance.Value;
14.            }
15.        }
16.        public void LogMessage(string message) {
17.            WriteLine("Message " + message);
18.        }
19.    }
20. }
```

3. Run the application and check the output.

We get the same output because lazy keyword created only one singleton class object and they are by default thread-safe that's why we do not get any error while invoking the methods accessing single instance parallelly.

Significance of Sealed Keyword in Singleton class

In the beginning of my last article on Singleton, I mentioned that we should use a sealed keyword before our singleton class. This was because we do not want our singleton class to be inherited. Sealed class helps us to restrict class inheritance but in singleton class, it does more than that because inheritance is anyways restricted in singleton class whether we use a sealed keyword or not in the class because all the constructors of singleton class are private which will never allow singleton class to get inherited. For e.g. create a derived class and try to make your singleton as a base class. Make the singleton as public class Singleton and add a new class in visual studio and call it DerivedClass, now make Singleton class as the base of this newly added class. We immediately see an error as follows.



It says that Singleton class is inaccessible due to its protection level. So why actually do we need sealed keyword when our purpose is already solved? We need a sealed keyword in case of nested classes. For e.g. consider a scenario where this derived class is the nested class of the singleton class and also inherits from the singleton class. As per OOP, it is perfectly possible. So our implementation becomes as follows,

```
    }
    2 references
    public void LogMessage(string message)
    {
      WriteLine("Message " + message);
    }
    0 references
    public class DerivedClass : Singleton
    {
    }
}
```

```
1. using System;
2. using static System.Console;
3. namespace Singleton {
4.     public class Singleton {
5.         static int instanceCounter = 0;
6.         private static readonly Lazy < Singleton > singleInstance = new Lazy < Singleton > (() => new Singleton()); //private static Singleton singleInstance
7.         private Singleton() {
8.             instanceCounter++;
9.             WriteLine("Instances created " + instanceCounter);
10.        }
11.        public static Singleton SingleInstance {
12.            get {
13.                return singleInstance.Value;
14.            }
15.        }
16.        public void LogMessage(string message) {
17.            WriteLine("Message " + message);
18.        }
19.        public class DerivedClass: Singleton {}
20.    }
21. }
```

Now try to access this derived class from the Main method of Program.cs.

```
0 references
static void Main(string[] args)
{
  Parallel.Invoke(() => LogManagerRequest(), () => LogEmployeeRequest());

  Singleton.DerivedClass derivedClass = new Singleton.DerivedClass();
  derivedClass.LogMessage("Log request for derived class invokation");

  ReadLine();
}
```

Since DerivedClass is a nested class we can call it via Singleton class and since it inherits from Singleton class it also has access to the LogMessage() method. Now run the application and check the output.

```
D:\Articles\Singleton\Singleton\Singleton\bin\Debug\Singleton.exe
Instances created 1
Message Request Message from Manager
Message Request Message from Employee
Instances created 2
Message Log request for derived class invocation
```

We clearly see that two instances of the Singleton class are created when we run the application. The first instance was created by the parallelInvoke's call to Log methods and second instance is created by the Derived class constructor as it implicitly called the base class constructor when the derived class instance was created. So in any case, whether it is a nested class or non-nested class, we have to restrict inheritance in a singleton class. Since private constructors do not help us in restricting inheritance in case of nested classes, we should use a sealed keyword to restrict inheritance. Now apply sealed keyword on Singleton class and check its nested derived class.

```
12 references
public sealed class Singleton
{
```

Derived class

```
2 references
public class DerivedClass : Singleton
{
}
```

class Singleton.Singleton.DerivedClass

'Singleton.DerivedClass': cannot derive from sealed type 'Singleton'

Now, when we have applied sealed on base singleton class, the derived class says that it cannot inherit from the sealed class, no matter if the derived is nested or not. So, we would never have two instances of singleton class as it would never be derived.

```
1. using System;
2. using static System.Console;
3. namespace Singleton {
4.     public sealed class Singleton {
5.         static int instanceCounter = 0;
6.         private static readonly Lazy < Singleton > singleInstance = new Lazy < Singleton > (() => new Singleton()); //private static Singleton singleInstance
7.         private Singleton() {
8.             instanceCounter++;
9.             WriteLine("Instances created " + instanceCounter);
10.        }
11.        public static Singleton SingleInstance {
12.            get {
13.                return singleInstance.Value;
14.            }
15.        }
16.        public void LogMessage(string message) {
17.            WriteLine("Message " + message);
18.        }
19.    }
20. }
```

Conclusion

In this article, we learned what is lazy initialization and what is eager initializations with practical examples. We also learned why it is necessary to make the singleton class sealed with the sealed keyword. In the next article, I'll try to explain the differences between static and singleton class and we will see where to use static and where to use singleton classes.

<<Click here for the previous article **Click here for the next article>>**