

Source: C# Corner ([www.c-sharpcorner.com](http://www.c-sharpcorner.com))

PRINT

---

## Article

---



### Singleton Design Pattern In C# - Part One

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

#### Introduction

I always wanted to write on Singleton design pattern in C#. Though there already are many posts available on Singleton design pattern, I'll try to cover this topic in the most simplistic and easy to understand way. This article will also talk about Static classes and the differences between singleton design pattern and static classes. Following is the link to next part of this article.

- [Singleton Design Pattern in C#: Part Two](#)

#### The Pattern Itself

The Singleton design pattern is a creational type of design pattern. We have distinct categories of design patterns, out of which creational is the one which deals with instance creation and initialization. This pattern helps a programmer to write a program with more flexibility of creating objects subjective to various cases, the example of which are Singleton, Factory, Abstract factory etc. Covering design pattern types are out of the scope of this article, so let's focus on Singleton design pattern.

A Singleton design pattern enables a developer to write code where only one instance of a class is created and the threads or other processes should refer to that single instance of the class. We, at a certain point in time, might be in a situation where we need to write a code that tells we only need one instance of a class and if some other class tries to create an object of that class, then the already instantiated object is shared to that class. One very common and suitable example is a log writer class. We might have a situation where we must maintain a single file for writing logs for requests coming to our .NET application from various clients like a mobile client, web client or any windows application client. In this case, there should be a class which takes this responsibility of writing the logs in a single file.

Since the requests come from multiple clients simultaneously, there should be a mechanism where only one request is logged at a time and on the other hand other requests should not be missed, instead should be taken care diligently to be logged, so that those requests when logged, do not override or conflict with the already logged requests. To achieve this, we can make a class singleton by following singleton pattern guidelines where a single thread-safe object is shared between all the requests which we'll discuss in detail in this article with practical examples.

#### Advantages of Singleton

Let's highlight the advantages of singleton class or pattern first before we jump into actual implementation. The first advantage that we can make out from the request logging example is that singleton takes care of concurrent access to a shared resource, which means if we are sharing a resource with multiple clients simultaneously, then it should be well taken care of. In our case, our log file is the shared resource and singleton makes sure, every client accesses it with no deadlock or conflict. The second advantage is that it only allows one instance of a class responsible for sharing the resource, which is shared across all the clients or applications in a controlled state.

#### Guidelines

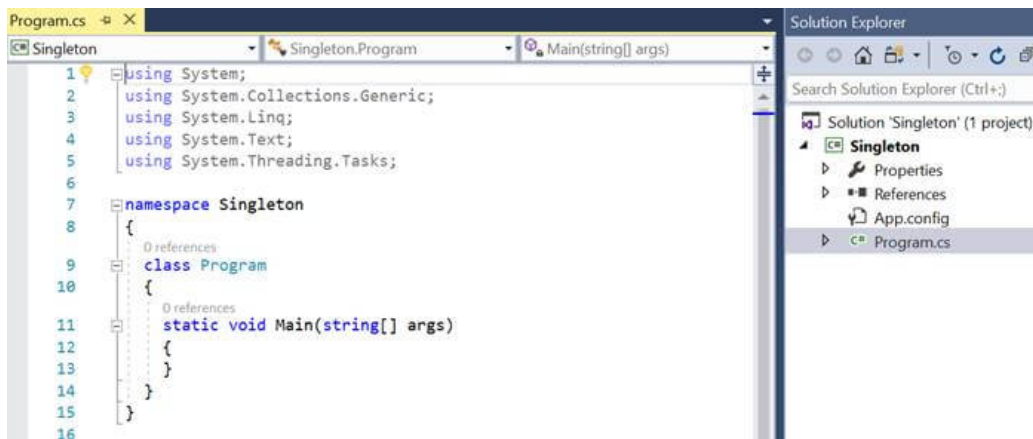
Every pattern is based on certain guidelines which should be followed while implementing the pattern. These guidelines help us build a robust pattern and each guideline has its significance in the pattern that we'll discuss which creating a singleton class. Stick to the following guidelines whenever you need to implement Singleton design pattern in C#.

1. Check that the implementation when done, creates only one instance of the class, and there should be only one point from where an instance is to be created.
2. The singleton's class constructors should be private so that no class can directly instantiate the singleton class.
3. There should be a static property/method that takes care of singleton class instantiation and that property should be shared across applications and is solely responsible for returning a singleton instance.
4. The C# singleton class should be sealed so that it could not be inherited by any other class, this is useful when we deal with nested class structure. We'll discuss this scenario as well later when we implement the singleton.

#### Basic Singleton Implementation

That's a lot of theory covered, now let's practically implement the singleton pattern. We'll cover this step by step.

1. Open the visual studio and create a console application named Singleton (you can choose whatever name you like).



2. Add a class named Singleton and add the following code to log a request message to it. For now, we are not actually logging into the file, but displaying the message at the console.

*Singleton.cs*

```

1. using static System.Console;
2. namespace Singleton
3. {
4.     class Singleton
5.     {
6.         public void LogMessage(string message)
7.         {
8.             WriteLine("Message " + message);
9.         }
10.    }
11. }
```

Let's also add a public constructor in the class with a variable to hold the counter of the number of objects created of this singleton class. We increment the instanceCounter variable whenever an instance of the Singleton class is created, so the best place to increment and print it is a constructor.

```

1. using static System.Console;
2.
3. namespace Singleton
4. {
5.     class Singleton
6.     {
7.         static int instanceCounter = 0;
8.         public Singleton()
9.         {
10.            instanceCounter++;
11.            WriteLine("Instances created " + instanceCounter);
12.        }
13.        public void LogMessage(string message)
14.        {
15.            WriteLine("Message " + message);
16.        }
17.    }
18. }
```

Note that we are not using any singleton guideline here and first we'll try to invoke this method from our main method i.e. in Program.cs class. So, go to Program.cs class and write following code for calling that LogMessage method of singleton class by creating an object of that class. We assume that two clients or two classes (Manager class and Employee class) are creating the objects of Singleton class to log their request message. Let's name those instances as "fromManager" and "fromEmployee".

*Program.cs*

```

1. using static System.Console;
2. namespace Singleton
3. {
4.     class Program
5.     {
6.         static void Main(string[] args)
7.         {
8.             Singleton fromManager = new Singleton();
9.             fromManager.LogMessage("Request Message from Manager");
10.
11.             Singleton fromEmployee = new Singleton();
12.             fromEmployee.LogMessage("Request Message from Employee");
13.
14.             ReadLine();
15.         }
16.     }
17. }
```

When we run the application, we see the output as following.

```
D:\Articles\Singleton\Singleton\Singleton\bin\Debug\Singleton.exe
Instances created 1
Message Request Message from Manager
Instances created 2
Message Request Message from Employee
```

We see here that we created two instances of our class named Singleton and got the method invoked by both the objects separately. When the first time fromManager instance was created, the counter was incremented by 1 and the second time at the time of fromEmployee instance creation, it is incremented again by 1 so 2 in total. Note that we have not yet implemented singleton guidelines and our aim is to restrict this multiple object creation.

3. If we again refer to the guidelines, it says that all the constructors of the singleton class should be private and there should be a property or method which takes responsibility of providing the instance of that singleton class. So, let's try that. Convert the public constructor in the Singleton class a private as following.

```
1. private Singleton()
2. {
3.     instanceCounter++;
4.     WriteLine("Instances created " + instanceCounter );
5. }
```

But when you go back to Program.cs class where we were making the instance we see the error as follows which is a compile-time error and appears as well when we compile the program at this point of time.

```
static void Main(string[] args)
{
    Singleton fromManager = new Singleton();
    fromManager.LogMessage("Request Message from Manager");

    Singleton fromEmployee = new Singleton();
    fromEmployee.LogMessage("Request

    ReadLine();
}
```

class Singleton.Singleton (+ 1 overload)

'Singleton.Singleton()' is inaccessible due to its protection level

It says that the constructor could not be accessed and that's because we have made it private. So, let's delegate this responsibility of serving objects to a static property in the Singleton class. Add a new property to the Singleton class

```
using static System.Console;

namespace Singleton
{
    8 references
    sealed class Singleton
    {
        static int instanceCounter = 0;
        private static Singleton singleInstance = null;
        1 reference
        private Singleton()
        {
            instanceCounter++;
            WriteLine("Instances created " + instanceCounter );
        }

        2 references
        public static Singleton SingleInstance
        {
            get
            {
                if (singleInstance == null)
                {
                    singleInstance = new Singleton();
                }
                return singleInstance;
            }
        }
    }
}
```

Following is the code,

```
1. using static System.Console;
2.
3. namespace Singleton
4. {
```

```

5. class Singleton
6. {
7.     static int instanceCounter = 0;
8.     private static Singleton singleInstance = null;
9.     private Singleton()
10.    {
11.        instanceCounter++;
12.        WriteLine("Instances created " + instanceCounter );
13.    }
14.
15.    public static Singleton SingleInstance
16.    {
17.        get
18.        {
19.            if (singleInstance == null)
20.            {
21.                singleInstance = new Singleton();
22.            }
23.            return singleInstance;
24.        }
25.    }
26.    public void LogMessage(string message)
27.    {
28.        WriteLine("Message " + message);
29.    }
30. }
31. }

```

If we go through the above code, we simply created a backing field named `singleInstance` and a public static property named `SingleInstance`. Whenever this property is accessed, it will instantiate the backing field with a new `Singleton` instance and return it to the client, but not every time, and to ensure that we check that if the `singleInstance` is null, then only it returns the new instance and not every time the property is accessed. For all other times, it should return the same instance as was initially created. Since the constructor of this class is now private, it could only be accessed from within the class members and not from outside the class. Now go to the `program.cs` class and access this property for getting the instance as we now cannot directly create a `Singleton` instance due to private constructor.

```

using static System.Console;
namespace Singleton
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Singleton fromManager = Singleton.SingleInstance;
            fromManager.LogMessage("Request Message from Manager");

            Singleton fromEmployee = Singleton.SingleInstance;
            fromEmployee.LogMessage("Request Message from Employee");

            ReadLine();
        }
    }
}

```

Following is the code.

```

1. using static System.Console;
2. namespace Singleton
3. {
4.     class Program
5.     {
6.         static void Main(string[] args)
7.         {
8.             Singleton fromManager = Singleton.SingleInstance;
9.             fromManager.LogMessage("Request Message from Manager");
10.
11.             Singleton fromEmployee = Singleton.SingleInstance;
12.             fromEmployee.LogMessage("Request Message from Employee");
13.
14.             ReadLine();
15.         }
16.     }
17. }

```

Now, when we run the application, we get following output.

```
Instances created 1
Message Request Message from Manager
Message Request Message from Employee
```

It clearly states that only one instance of the Singleton class was created but our methods were called distinctly for both the callers. This is because at the first time the instance was created but the second time when the property was accessed by fromEmployee the already created object was returned.

So, no doubt that we have implemented the single design pattern, and changed the object creation strategy of this Singleton class, but this is a very basic implementation of Singleton design pattern and it still does not take care of deadlock situations and accessing class in a multithreaded environment. Let's see how we can make this class thread-safe as well. Make the singleton class sealed before we proceed. We'll discuss why we made the class sealed later.

## Thread-Safe Singleton Implementation

### Problem

Our basic level implementation only would work in a single-threaded system because our instance creation is lazily initialized which means we only create the instance when the SingletonInstance property is invoked. Suppose there is a situation where two threads try to access the property at the same time, in that case, it could be a situation that both the threads hit the null check at the same time and get access to new instance creation because they find the instance variable still null. Let's test this scenario in our current Singleton implementation.

Go to the Program.cs class and create two methods named LogEmployeeRequest and LogManagersRequest and move the logging code for both the instances to these methods as shown below.

```
using static System.Console;
namespace Singleton
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            LogManagerRequest();
            LogEmployeeRequest();
            ReadLine();
        }

        1 reference
        private static void LogManagerRequest()
        {
            Singleton fromManager = Singleton.SingletonInstance;
            fromManager.LogMessage("Request Message from Manager");
        }

        1 reference
        private static void LogEmployeeRequest()
        {
            Singleton fromEmployee = Singleton.SingletonInstance;
            fromEmployee.LogMessage("Request Message from Employee");
        }
    }
}
```

Now, let's try to invoke these methods in parallel using Parallel.Invoke method as shown below. This method can invoke multiple methods parallelly and so we would get into a situation where both methods claim Singleton instance at the same time. The parallel is the class under System.Threading.Tasks namespace.

```

using System.Threading.Tasks;
using static System.Console;
namespace Singleton
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Parallel.Invoke(() => LogManagerRequest(), () => LogEmployeeRequest());
            ReadLine();
        }

        1 reference
        private static void LogManagerRequest()
        {
            Singleton fromManager = Singleton.SingleInstance;
            fromManager.LogMessage("Request Message from Manager");
        }

        1 reference
        private static void LogEmployeeRequest()
        {
            Singleton fromEmployee = Singleton.SingleInstance;
            fromEmployee.LogMessage("Request Message from Employee");
        }
    }
}

```

Following is the code.

```

1. using System.Threading.Tasks;
2. using static System.Console;
3. namespace Singleton
4. {
5.     class Program
6.     {
7.         static void Main(string[] args)
8.         {
9.             Parallel.Invoke(() => LogManagerRequest(), () => LogEmployeeRequest());
10.            ReadLine();
11.        }
12.
13.        private static void LogManagerRequest()
14.        {
15.            Singleton fromManager = Singleton.SingleInstance;
16.            fromManager.LogMessage("Request Message from Manager");
17.        }
18.        private static void LogEmployeeRequest()
19.        {
20.            Singleton fromEmployee = Singleton.SingleInstance;
21.            fromEmployee.LogMessage("Request Message from Employee");
22.        }
23.    }
24. }

```

Now, when we run the application, we get the following output.

```

D:\Articles\Singleton\Singleton\Singleton\bin\Debug\Singleton.exe
Instances created 1
Instances created 2
Message Request Message from Manager
Message Request Message from Employee

```

The above output clearly states that we ended up creating two instances of Singleton class as our constructor was called twice. That's because both the methods executed at the same time. Now to overcome this situation we can further enhance our Singleton class.

### Solution

One of the ways to overcome this situation is to use locks. We can use lock over an object if any thread tries to access the instance and in that case, other thread waits until the lock is released. Let's implement this. Update your Singleton class in the following manner.

```

{
    static int instanceCounter = 0;
    private static Singleton singleInstance = null;
    private static readonly object lockObject = new object();
    1 reference
    private Singleton()
    {
        instanceCounter++;
        WriteLine("Instances created " + instanceCounter );
    }

    2 references
    public static Singleton SingleInstance
    {
        get
        {
            lock (lockObject)
            {
                if (singleInstance == null)
                {
                    singleInstance = new Singleton();
                }
            }

            return singleInstance;
        }
    }
}

```

Following is the code.


```

1. using static System.Console;
2.
3. namespace Singleton
4. {
5.     sealed class Singleton
6.     {
7.         static int instanceCounter = 0;
8.         private static Singleton singleInstance = null;
9.         private static readonly object lockObject = new object();
10.        private Singleton()
11.        {
12.            instanceCounter++;
13.            WriteLine("Instances created " + instanceCounter );
14.        }
15.
16.        public static Singleton SingleInstance
17.        {
18.            get
19.            {
20.                lock (lockObject)
21.                {
22.                    if (singleInstance == null)
23.                    {
24.                        singleInstance = new Singleton();
25.                    }
26.                }
27.            }
28.            return singleInstance;
29.        }
30.    }
31.    public void LogMessage(string message)
32.    {
33.        WriteLine("Message " + message);
34.    }
35. }

```

```
36. }
```

In the above-mentioned source code, we created a private static read-only object type variable and initialized it. Then in the Singleton property, we wrapped the code of instance creation under the lock, so that one thread can enter the code at a time and the other thread waits until the first thread finishes its execution. Now, let's run the application and check the output. Compile and Run.

 D:\Articles\Singleton\Singleton\Singleton\bin\Debug\Singleton.exe

```
Instances created 1
Message Request Message from Employee
Message Request Message from Manager
```

So, now only one instance of Singleton class is created, that means our lock works fine. We still have a problem with the current implementation. The problem is that our lock object code will be called every time the property of Singleton is accessed which may incur a huge performance cost on the application as locks are quite expensive when we want good performance in our application. So, we can restrict this every time lock code access by wrapping it under the condition that it could only be accessed if the singleton backing field is null. Therefore, our code for the property is like as shown below.

```
1. public static Singleton SingletonInstance
2. {
3.     get
4.     {
5.         if (singletonInstance == null)
6.         {
7.             lock (lockObject)
8.             {
9.                 if (singletonInstance == null)
10.                {
11.                    singletonInstance = new Singleton();
12.                }
13.            }
14.        }
15.    }
16.    return singletonInstance;
17. }
18. }
```

Now when we run the application, the lock code would not be executed every time but only for the first time when it is accessed because at the second time it will not find the singletonInstance field as null. The complete class code is as following.

```
1. using static System.Console;
2.
3. namespace Singleton
4. {
5.     sealed class Singleton
6.     {
7.         static int instanceCounter = 0;
8.         private static Singleton singletonInstance = null;
9.         private static readonly object lockObject = new object();
10.        private Singleton()
11.        {
12.            instanceCounter++;
13.            WriteLine("Instances created " + instanceCounter );
14.        }
15.
16.        public static Singleton SingletonInstance
17.        {
18.            get
19.            {
20.                if (singletonInstance == null)
21.                {
22.                    lock (lockObject)
23.                    {
24.                        if (singletonInstance == null)
25.                        {
26.                            singletonInstance = new Singleton();
27.                        }
28.                    }
29.                }
30.            }
31.            return singletonInstance;
32.        }
33.    }
34.    public void LogMessage(string message)
35.    {
36.        WriteLine("Message " + message);
37.    }
38. }
39. }
```



We call this null instance check locking as “*double check locking*”, which is often asked in the interviews.

## Conclusion

In this article, we discussed what is Singleton design pattern, when it's needed and what problems it 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. For the sake of not making the article too long, I have divided the learning Singleton topic in several parts. Please refer [next part](#) of this article where we discuss lazy initialization, why to make singleton class a sealed class and what are the differences between singleton and static class.

[Click here for next Article>>](#)

---

Thank you for using C# Corner