Singleton VS. Static Classes

Many of us have used static and singleton classes in our projects. Sometimes people ask "Why do you use a Singleton class if a Static class serves the purpose?" Or someone might have asked you at least once "What is the difference between Singleton and Static classes and when do you use each one in your program?" I have been asked many times by friends and in some interviews (most of the times) and so thought of sharing whatever is in my mind about this.

Static and Singleton are very different in their usage and implementation. So we need to wisely choose either of these two in our projects.

Let us discuss more about the Singleton class first.

Singleton is a design pattern that makes sure that your application creates only one instance of the class anytime. It is highly efficient and very graceful. Singletons have a static property that you must access to get the object reference. So you are not instantiating an object such as in the manner we do for a normal class.

We go through the following code and learn more about normal class usage, static class usage and about the singleton class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace SingletonVsStaticClass
{
    class SomeClass
    {
        public int ABC;
    }

    public sealed class SingletonSampleClass
    {
        static  SingletonSampleClass _instance;

        public static SingletonSampleClass Instance
        {
            get { return _instance ?? (_instance = new SingletonSampleClass()); }
        }
        private SingletonSampleClass() { }

        public string Message { get; set; }
    }
    static public class StaticSampleClass
    {
        private static readonly int SomeVariable;
        //Static constructor is executed only once when the type is first used.
        //All classes can have static constructors, not just static classes.
        static StaticSampleClass()
        {
            SomeVariable = 1;
            //Do the required things
        }
        public static string ShowValue()
        {
            return string.Format("The value of someVariable is {0}", SomeVariable);
        }
        public static string Message { get; set; }
    }
    class Program
    {

        static void Main(string[] args)
        {
            //Normal class instantiation and usage
            var someClass = new SomeClass {ABC = 5};
            Console.WriteLine("Normal class usage: "+someClass.ABC);

            //Static Class instantiation
            string returnValue = StaticSampleClass.ShowValue();
            Console.WriteLine("Static class usage: " + returnValue);

            //Singleton class instantiation
            var singletonSampleClass = SingletonSampleClass.Instance;
            singletonSampleClass.Message = "Hello";
            Console.WriteLine("Singleton class usage: " + singletonSampleClass.Message);

            //Test the instances if are equal
            var anotherSingletonSampleClass = SingletonSampleClass.Instance;
            Console.WriteLine("Checking if both instances are same: "+ singletonSampleClass.E
quals(anotherSingletonSampleClass) );

            Console.Read();
        }
    }
```

```
}
```

```
[illegible]
```

You saw how to use properties and methods for various types of classes. For a normal class, we need to create an object first to use its properties and methods. For static classes we do not need to instantiate and we access properties and methods by class name. For singleton classes, we create an instance using its static property and at any time it creates a single instance of a class. You should have checked in the above example that even when you try to create multiple instances of a singleton class, it returns the same instance as was created the first time.

When shall we use singleton class and when to go for static classes?

Static classes are basically used when you want to store a single instance, data which should be accessed globally throughout your application. The class will be initialized at any time but mostly it is initialized lazily. Lazy initialization means it is initialized at the last possible moment of time. There is a disadvantage of using static classes. You never can change how it behaves after the class is decorated with the static keyword.

Singleton Class instance can be passed as a parameter to another method whereas static class cannot

The Singleton class does not require you to use the static keyword everywhere. Static class objects cannot be passed as parameters to other methods whereas we can pass instances of a singleton as a parameter to another method.

For example we can modify our normal class to have a method which takes a singleton class instance as a parameter. We cannot do this with static classes.

```
class SomeClass
   {
      public int ABC;

      public void SingletonMethod(SingletonSampleClass singletonSampleClass)
      {

      }
   }
```

```
var someClass = new SomeClass {ABC = 5};
someClass.SingletonMethod(anotherSingletonSampleClass);
```

Singleton classes support Interface inheritance whereas a static class cannot implement an interface:

Let us try to create an interface and see if we can make our 3 classes to implement it; see:

```
 public interface IMyInterface
    {
        void MyInterfaceMethod();
    }
    class SomeClass : IMyInterface
    {
        public int ABC;

        public void SingletonMethod(SingletonSampleClass singletonSampleClass)
        {

        }

        public void MyInterfaceMethod()
        {
            throw new NotImplementedException();
        }
    }

    public sealed class SingletonSampleClass : IMyInterface
    {
        static  SingletonSampleClass _instance;

        public static SingletonSampleClass Instance
        {
            get { return _instance ?? (_instance = new SingletonSampleClass()); }
        }
        private SingletonSampleClass() { }

        public string Message { get; set; }

        public void MyInterfaceMethod()
        {
            throw new NotImplementedException();
        }
    }

    static public class StaticSampleClass:IMyInterface
    {
        private static readonly int SomeVariable;
        //Static constructor is executed only once when the type is first used.
        //All classes can have static constructors, not just static classes.
        static StaticSampleClass()
        {
            SomeVariable = 1;
            //Do the required things
        }
        public static string ShowValue()
        {
            return string.Format("The value of someVariable is {0}", SomeVariable);
        }
        public static string Message { get; set; }
    }
```

When we are tryg to make the static class implement the interface, it gives the following exception:

SingletonVsStaticClass.StaticSampleClass: static classes cannot implement interfaces

This is because static classes cannot implement interfaces.

Since singleton class supports interface implementation, we can reuse our singleton for any number of implementations of interface confirming objects.

```
CustomMethod((IMyInterface)singletonSampleClass);
```

Your code will be more flexible if you are using a singleton class and another advantage is that the code that uses the singleton does not need to know if it is a singleton object or a transient object. Using a static means you have to call static methods explicitly. Static classes also fail during dependency injection.
There is a general problem that I have read somewhere in one of the articles. Consider the possibility that our application has a global shared object and tomorrow you decide that you must make more than one instance of the class.
If you use a singleton class then all you have to do is make the constructor public. This is not so simple with static classes.

During my initial days of my career, I had created a static class and set a property for a

page heading. So each page, while loading, sets the page heading and uses the same page heading for consecutive pages throughout the web application. Based on the user's selection at login, each user will have a different heading set and the same should be displayed until he logs out. There are two issues seen in this case. One is the heading changes for all the users when one someone logins with a different option. And also, until the server is restarted or all the applications closed, the static variables cannot be reset. This is a problem in web applications. So, we need to be careful with static variables and static methods in web applications.
The advantage of using a static class is the compiler makes sure that no instance methods are accidentally added. The compiler guarantees that instances of the class cannot be created.

A singleton class has a private constructor that prevents the class from being instantiated. A singleton class can have instance members while a static class cannot.

Thread safe for singleton class instantiation

The following is beautifully explained in the MSDN and I am pasting the content from the MSDN site here. You can access the content from the URL http://msdn.microsoft.com/en-us/library/ff650316.aspx

Even though Singleton is a comparatively simple pattern, there are various tradeoffs and options, depending upon the implementation. The following is a series of implementation strategies with a discussion of their strengths and weaknesses.

Singleton

The following implementation of the Singleton design pattern follows the solution presented in Design Patterns: Elements of Reusable Object-Oriented Software [Gamma95] but modifies it to take advantage of language features available in C#, such as properties:

```
using System;

public class Singleton
{
  private static Singleton instance;

  private Singleton() {}

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

This implementation has two main advantages:

- Because the instance is created inside the Instance property method, the class can exercise additional functionality (for example, instantiating a subclass), even though it may introduce unwelcome dependencies.
- The instantiation is not performed until an object asks for an instance; this approach is referred to as lazy instantiation. Lazy instantiation avoids instantiating unnecessary singletons when the application starts.

The main disadvantage of this implementation, however, is that it is not safe for multithreaded environments. If separate threads of execution enter the Instance property method at the same time, more than one instance of the Singleton object may be created. Each thread could execute the following statement and decide that a new instance has to be created:

```
if (instance == null)
```

Various approaches solve this problem. One approach is to use an idiom referred to as Double-Check Locking [Lea99]. However, C# in combination with the Common Language Runtime provides a static initializationapproach, which circumvents these issues without requiring the developer to explicitly code for thread safety.

Static Initialization

One of the reasons Design Patterns [Gamma95] avoided static initialization is because the C++ specification left some ambiguity around the initialization order of static variables. Fortunately, the .NET Framework resolves this ambiguity through its handling of variable initialization:

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

   private Singleton(){}

   public static Singleton Instance
```

```
  {
    get
    {
      return instance;
    }
  }
}
```

In this strategy, the instance is created the first time any member of the class is referenced. The Common Language Runtime takes care of the variable initialization. The class is marked sealed to prevent derivation, which could add instances. For a discussion of the pros and cons of marking a class sealed, see [Sells03]. In addition, the variable is marked readonly, which means that it can be assigned only during static initialization (which is shown here) or in a class constructor.

This implementation is similar to the preceding example, except that it relies on the Common Language Runtime to initialize the variable. It still addresses the two basic problems that the Singleton pattern is trying to solve: global access and instantiation control. The public static property provides a global access point to the instance. Also, because the constructor is private, the Singleton class cannot be instantiated outside of the class itself; therefore, the variable refers to the only instance that can exist in the system.

Because the Singleton instance is referenced by a private static member variable, the instantiation does not occur until the class is first referenced by a call to the Instance property. This solution therefore implements a form of the lazy instantiation property, as in the Design Patterns form of Singleton.
The only potential downside of this approach is that you have less control over the mechanics of the instantiation. In the Design Patterns form, you were able to use a nondefault constructor or perform other tasks before the instantiation. Because the .NET Framework performs the initialization in this solution, you do not have these options. In most cases, static initialization is the preferred approach for implementing a Singleton in .NET.

Multithreaded Singleton

Static initialization is suitable for most situations. When your application must delay the instantiation, use a non-default constructor or perform other tasks before the instantiation, and work in a multithreaded environment, you need a different solution. Cases do exist, however, in which you cannot rely on the Common Language Runtime to ensure thread safety, as in the Static Initialization example. In such cases, you must use specific language capabilities to ensure that only one instance of the object is created in the presence of multiple threads. One of the more common solutions is to use the Double-Check Locking [Lea99] idiom to keep separate threads from creating new instances of the singleton at the same time.

Note: The Common Language Runtime resolves issues related to using Double-Check Locking that are common in other environments. For more information about these issues, see "The 'Double-Checked Locking Is Broken' Declaration," on the University of Maryland, Department of Computer Science Web site, athttp://www.cs.umd.edu/%7Epugh/java/memoryModel/DoubleCheckedLocking.html.

The following implementation allows only a single thread to enter the critical area, which the lock block identifies, when no instance of Singleton has yet been created:

```
using System;

public sealed class Singleton
{
   private static volatile Singleton instance;
   private static object syncRoot = new Object();

   private Singleton() {}

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

         return instance;
      }
   }
}
```

This approach ensures that only one instance is created and only when the instance is needed. Also, the variable is declared to be volatile to ensure that assignment to the instance variable completes before the instance variable can be accessed. Lastly, this approach uses a syncRoot instance to lock on, rather than locking on the type itself, to avoid deadlocks.

This double-check locking approach solves the thread concurrency problems while avoiding an exclusive lock in every call to the Instance property method. It also allows you to delay instantiation until the object is first accessed. In practice, an application rarely requires this type of implementation. In most cases, the static initialization approach is sufficient.

Resulting Context

Implementing Singleton in C# results in the following benefits and liabilities:

Benefits

- The static initialization approach is possible because the .NET Framework explicitly defines how and when static variable initialization occurs.
- The Double-Check Locking idiom described earlier in "Multithreaded Singleton" is implemented correctly in the Common Language Runtime.

Liabilities

If your multithreaded application requires explicit initialization, you have to take precautions to avoid threading issues.

http://www.c-sharpcorner.com/UploadFile/akkiraju/singleton-vs-static-classes/