MICROSOFT ARCHITECT
By Joydip Kanjilal, Columnist, InfoWorld
JAN 4, 2021 3:00 AM PST

Singleton vs. static classes in C#

Understand the differences between a singleton class and a static class and when to use which in your applications.

When developing applications in .NET Core, you might often need a single, shared instance of a class. Typical use cases would be an instance of LogManager, StateManager, etc. You can either use a singleton class or a static class to achieve this. The decision on which to choose—singleton or static—depends on several factors. This article explains singleton classes and static classes and when we should we use one versus the other.

To work with the code examples provided in this article, you should have Visual Studio 2019 installed in your system. If you don't already have a copy, you can download Visual Studio 2019 here. You can download .NET 5.0 from here.

[Also on InfoWorld: The most valuable software developer skills in 2020]

Create an ASP.NET Core 5 MVC project in Visual Studio 2019

First off, let's create an ASP.NET Core project in Visual Studio 2019. Following these steps should create a new ASP.NET 5 project in Visual Studio 2019.

1. Launch the Visual Studio IDE.
2. Click on "Create new project."
3. In the "Create new project" window, select "ASP.NET Core Web App (Model-View-Controller)" from the list of templates displayed.
4. Click Next.

5. In the “Configure your new project” window, specify the name and location for the new project.
6. Optionally check the “Place solution and project in the same directory” check box, depending on your preferences.
7. Click Next.
8. In the “Additional Information” window shown next, select .NET 5.0 as the target framework from the drop-down list at the top.
9. Ensure that the check boxes “Enable Docker,” “Configure for HTTPS,” and “Enable Razor runtime compilation” are unchecked as we won’t be using any of those features here.
10. Ensure that Authentication is set to “None” as we won’t be using authentication either.
11. Click Create.

Singleton class or static class? How to choose

We’ll compare and contrast a singleton class and a static class based on the following points:

- Dependency injection
- Memory management
- Extensibility
- Testability

In the sections that follow, we’ll implement two classes — a static class named StaticLogger and a singleton class named SingletonLogger. Both of these classes provide a Log method that can be used to log data to a specific log target. In both examples below I have omitted the necessary code for logging the data for brevity. (You can read more about static classes and static class members in C# in my previous article here.)

Create a static logger class in C#

Static classes cannot be instantiated or extended. They are abstract and sealed implicitly. To declare a class as static, you should mark it with the static keyword in the class declaration. The static keyword allows you to define both static classes and static members.

The following code snippet illustrates a static class.


```
public static class StaticLogger
    {
        private static readonly object lockObj = new object();
        public static void Log(string message)
        {
            //Write code here to log data.
        }
    }
```

Typically, static classes are used to implement helper or utility classes. They generally contain a collection of some reusable methods and properties.

Create a singleton logger class in C#

A singleton class, an implementation of the singleton design pattern, is a class of which only a single instance can exist.

The following code listing illustrates a minimalistic implementation of a singleton class. The static Instance property can be used to invoke the members of the singleton class.

```
public sealed class SingletonLogger
 {
     private static SingletonLogger instance;
     private static object lockObj = new Object();
     private SingletonLogger () { }
     public static SingletonLogger Instance
     {
         get
         {
             lock (lockObj)
             {
                 if (instance == null)
                     instance = new SingletonLogger();
             }
             return instance;
         }
     }
     public void Log(string message)
     {
         //Write code here to log data.
     }
 }
```

Singleton vs. static classes and dependency injection

ASP.NET Core 5 MVC has built-in support for dependency injection. When working in ASP.NET Core MVC you can add services to the container in the ConfigureServices method of the Startup class. These services are then made available to other classes in the application using dependency injection. You can then take advantage of dependency injection in the controller or other classes to use the instances injected.

Assuming you have a non-static class called FileLogger that implements an interface called ILogger, you can use the following code snippet to add a service to the container with a singleton lifetime.

```
public void ConfigureServices(IServiceCollection services)
{
      services.AddControllersWithViews();
      services.AddSingleton<ILogger, FileLogger>();
}
```

You need not write any code for implementing the singleton pattern yourself—the ASP.NET Core 5 MVC runtime will take care of it.

By contrast, if you attempt to inject a static class the same way you injected an instance of a non-static class in the preceding example, you will receive an error. This is primarily because static classes don't have any instances and hence cannot be used as type arguments.

The following code snippet illustrates this.

```
public void ConfigureServices(IServiceCollection services)
{
   services.AddControllersWithViews();
   services.AddSingleton<StaticLogger>();
}
```

When you compile the application, you will be presented with the following error.

```
static types cannot be used as type arguments
```

You can use dependency injection in a static class using method or property injection. However, you cannot use constructor injection in a static class because the constructor of a static class cannot accept any parameters.

Singleton vs. static classes and memory management

Static objects and static classes are stored in a region of the managed heap known as the high frequency heap. Objects stored in the high frequency heap are released only when the application unloads. The single instance of a singleton class is static and hence an instance of the singleton class is stored in the high frequency heap. Thus the memory occupied by an instance of the singleton class is released when the application unloads or is terminated.

However, unlike a static class that can have only static objects, a singleton class can have both static and non-static objects. Hence from the perspective of memory management, you can take advantage of garbage collection for the managed objects when you're using a singleton class.

Singleton vs. static classes and extensibility

A singleton class typically contains a private constructor and is marked as sealed to indicate that it can neither be instantiated nor inherited any further. Hence, you can extend a singleton class only if you have a non-private constructor in the singleton class as shown in the code snippet given below.

```
public class SingletonLogger
    {
        protected SingletonLogger() { }
        //Other members
    }
```

You can now extend the singleton class as shown below.

```
public class LogManager : SingletonLogger
    {
        //Write your implementation here
    }
```

Likewise, you cannot inherit a static class and override its methods. Moreover, while you can have extension methods in a singleton class, a static class cannot have extension methods.

Singleton vs. static classes and flexibility

While you can clone an instance of a singleton class, the same is not possible in a static class. You can have the Dispose method in a singleton class but not in a static class. You cannot instantiate a static class, so it cannot be used wherever the “this” reference is required (an indexer, a method parameter). Any member of a static class such as a constructor, a field, a property, or an event is static. Whenever you use a static class, you don’t have any control over when the static constructor is called.

Singleton classes can be designed to load lazily, while static classes can be loaded only eagerly. Lazy initialization is a technique that defers object creation, i.e., an instance of a class can be loaded on demand so as to improve the performance of an application and reduce memory requirements.

Singleton vs. static classes and testability

While testing a singleton class is easy, the same cannot be said for a static class. It is extremely difficult (if not impossible) to mock a static class. The tests executed on a static class might affect one another because those tests don't execute on different instances.

Note that the static methods pertaining to a static class are not untestable in themselves. A static method that holds no state or doesn't change state can be unit tested. As long as the method and its dependencies are idempotent, the method can be unit tested. The problems arise when the static method calls other methods or when the object being tested calls the static method.

To sum up, a static class is one that can't have any instances and contains only static members, i.e., members that are not associated with a particular instance. A static class represents a unit of organization for a set of methods that are not associated with any particular instance.

A static class is a good choice when you only need a utility class that contains several utility methods—you don't need an instance in such cases. This results in a simple implementation and improves the application's performance as well, because you don't have any instances for such a class.

The singleton pattern can be used for designing classes for which you need just one instance. Typical examples include manager classes for use in logging, caching, thread pools, etc. A singleton class would also be a good choice when you want to manage a shared resource such as a printer spooler. You should have a single instance for such a purpose to avoid conflicting requests for the same resource.

How to do more in C#:

- How to log data to the Windows Event Log in C#
- How to use ArrayPool and MemoryPool in C#
- How to use the Buffer class in C#
- How to use HashSet in C#
- How to use named and optional parameters in C#
- How to benchmark C# code using BenchmarkDotNet
- How to use fluent interfaces and method chaining in C#

- How to unit test static methods in C#

- How to refactor God objects in C#
- How to use ValueTask in C#
- How to use immutability in C
- How to use const, readonly, and static in C#
- How to use data annotations in C#
- How to work with GUIDs in C# 8
- When to use an abstract class vs. interface in C#
- How to work with AutoMapper in C#
- How to use lambda expressions in C#
- How to work with Action, Func, and Predicate delegates in C#
- How to work with delegates in C#
- How to implement a simple logger in C#
- How to work with attributes in C#
- How to work with log4net in C#
- How to implement the repository design pattern in C#
- How to work with reflection in C#
- How to work with filesystemwatcher in C#
- How to perform lazy initialization in C#
- How to work with MSMQ in C#
- How to work with extension methods in C#
- How to us lambda expressions in C#
- When to use the volatile keyword in C#
- How to use the yield keyword in C#
- How to implement polymorphism in C#
- How to build your own task scheduler in C#
- How to work with RabbitMQ in C#
- How to work with a tuple in C#

- Exploring virtual and abstract methods in C#
- How to use the Dapper ORM in C#
- How to use the flyweight design pattern in C#

Joydip Kanjilal is a Microsoft MVP in ASP.Net, as well as a speaker and author of several books and articles. He has more than 20 years of experience in IT including more than 16 years in Microsoft .Net and related technologies.

Follow

Red Hat Enterprise Linux takes aim at edge computing

The shifting market for PostgreSQL

Are industry clouds an opportunity or a distraction?

Make life easy with ssh_config

6 reasons to switch to managed Kubernetes

What is a computational storage drive? Much-needed

When to use Task.WaitAll vs. Task.WhenAll in .NET

6 ways Alibaba Cloud challenges AWS, Azure, and GCP