CHAPTER 1

Singleton Pattern

This chapter covers the Singleton pattern.

GoF Definition

Ensure a class has only one instance, and provide a global point of access to it.

Concept

A particular class should have only one instance. You can use this instance whenever you need it and therefore avoid creating unnecessary objects.

Real-Life Example

Suppose you are a member of a sports team and your team is participating in a tournament. When your team plays against another team, as per the rules of the game, the captains of the two sides must have a coin toss. If your team does not have a captain, you need to elect someone to be the captain first. Your team must have one and only one captain.

Computer World Example

In some software systems, you may decide to maintain only one file system so that you can use it for the centralized management of resources.

V. Sarcar, *Design Patterns in C#*, https://doi.org/10.1007/978-1-4842-3640-6_1

Illustration

These are the key characteristics in the following implementation:

- The constructor is private in this example. So, you cannot instantiate in a normal fashion (using `new`).
- Before you attempt to create an instance of a class, you check whether you already have an available copy. If you do not have any such copy, you create it; otherwise, you simply reuse the existing copy.

Class Diagram

Figure 1-1 shows the class diagram for the illustration of the Singleton pattern.

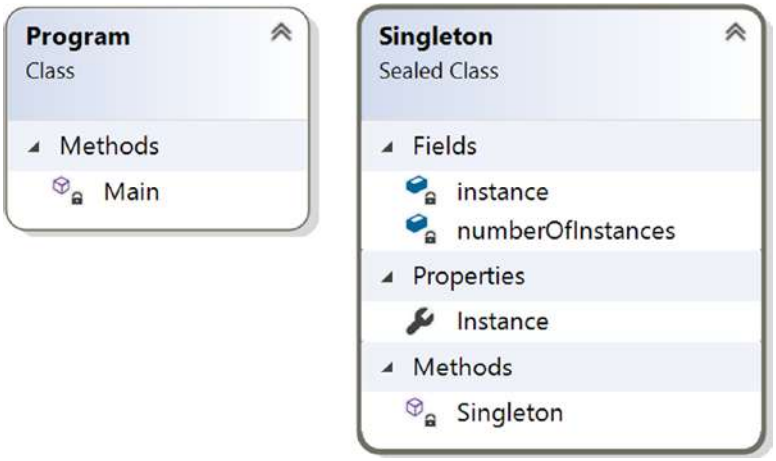

Figure 1-1. *Class diagram*

Solution Explorer View

Figure 1-2 shows the high-level structure of the parts of the program.

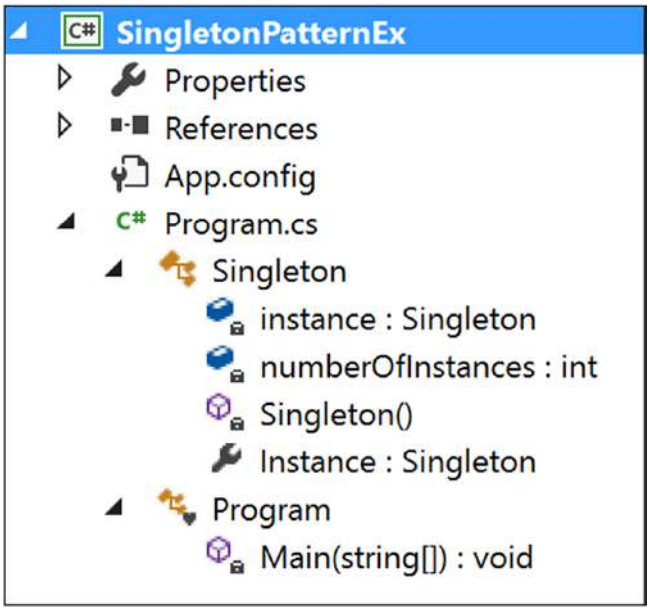

Figure 1-2. *Solution Explorer View*

Discussion

This simple example illustrates the concept of the Singleton pattern. This approach is known as *static initialization*.

Initially the C++ specification had some ambiguity about the initialization order of static variables (remember that the origin of C# is closely tied with C and C++), but the .NET Framework resolved these issues.

The following are the notable characteristics of this approach:

- The Common Language Runtime (CLR) takes care of the variable initialization process.
- You create an instance when any member of the class is referenced.

- The `public static` member ensures a global point of access. It confirms that the instantiation process will not start until you invoke the `Instance` property of the class (in other words, it supports lazy instantiation). The `sealed` keyword prevents the further derivation of the class (so that its subclass cannot misuse it), and `readonly` ensures that the assignment process takes place during the static initialization.
- The constructor is private. So, you cannot instantiate the `Singleton` class inside `Main()`. This will help you refer to the one instance that can exist in the system.

Implementation

Here is the implementation of the example:

```
using System;

namespace SingletonPatternEx
{
    public sealed class Singleton
    {
        private static readonly Singleton instance=new Singleton();
        private int numberOfInstances = 0;
        //Private constructor is used to prevent
        //creation of instances with 'new' keyword outside this class
        private Singleton()
        {
         Console.WriteLine("Instantiating inside the private constructor.");
         numberOfInstances++;
         Console.WriteLine("Number of instances ={0}", numberOfInstances);
        }
        public static Singleton Instance
        {
            get
            {
                Console.WriteLine("We already have an instance now.Use it.");
```

```
            return instance;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***Singleton Pattern Demo***\n");
        //Console.WriteLine(Singleton.MyInt);
        // Private Constructor.So,we cannot use 'new' keyword.
        Console.WriteLine("Trying to create instance s1.");
        Singleton s1 = Singleton.Instance;
        Console.WriteLine("Trying to create instance s2.");
        Singleton s2 = Singleton.Instance;
        if (s1 == s2)
        {
            Console.WriteLine("Only one instance exists.");
        }
        else
        {
            Console.WriteLine("Different instances exist.");
        }
        Console.Read();
    }
}
}
```

Output

Here is the output of the example:

```
***Singleton Pattern Demo***

Trying to create instance s1.
Instantiating inside the private constructor.
```

```
Number of instances =1
We already have an instance now.Use it.
Trying to create instance s2.
We already have an instance now.Use it.
Only one instance exists.
```

Challenges

Consider the following code. Suppose you have added one more line of code (shown in bold) in the Singleton class.

```
public sealed class Singleton
    {
        private static readonly Singleton instance = new Singleton();
        private int numberOfInstances = 0;
        //Private constructor is used to prevent
        //creation of instances with 'new' keyword outside this class
        private Singleton()
        {
           Console.WriteLine("Instantiating inside the private constructor.");
            numberOfInstances++;
           Console.WriteLine("Number of instances ={0}", numberOfInstances);
        }
        public static Singleton Instance
        {
            get
            {
                Console.WriteLine("We already have an instance now.Use it.");
                return instance;
            }
        }
        public static int MyInt = 25;
    }
```

And suppose, your `Main()` method looks like this:

```
class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Singleton Pattern Demo***\n");
            Console.WriteLine(Singleton.MyInt);
            Console.Read();
        }
    }
```

Now, if you execute the program, you will see following output:

```
Number of instances =1
***Singleton Pattern Demo***

25
```

This illustrates the downside of this approach. Specifically, inside the `Main()` method, you tried to use the static variable `MyInt`, but your application still created an instance of the `Singleton` class. In other words, with this approach, you have less control over the instantiation process, which starts whenever you refer to any static member of the class.

However, in most cases, you do not care about this drawback. You can tolerate it because you know that it is a one-time activity and the process will not be repeated, so this approach is widely used in .NET applications.

Q&A Session

1. **Why are you complicating stuff? You can simply write your `Singleton` class as follows:**

```
public class Singleton
    {
        private static Singleton instance;

        private Singleton() { }
```

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

Answer:

This approach can work in a single-threaded environment. But consider a multithreaded environment. In a multithreaded environment, suppose two (or more) threads try to evaluate this:

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

If they see that the instance has not been created yet, each of them will try to create a new instance. As a result, you may end up with multiple instances of the class.

2. **Are there any alternative approaches for modeling Singleton design patterns?**

   Answer:

   There are many approaches. Each of them has pros and cons. I'll discuss one approach called *double checked locking*. MSDN outlines the approach as shown here:

```
//Double checked locking
    using System;

    public sealed class Singleton
```

```
{
    //We are using volatile to ensure that
    //assignment to the instance variable finishes before it's
    //access.
    private static volatile Singleton instance;
    private static object lockObject = new Object();

    private Singleton() { }

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

This approach can help you create the instances when they are really needed. But you must remember that, in general, the locking mechanism is expensive.

If you are further interested in Singleton patterns, you can refer to http://csharpindepth.com/Articles/General/Singleton.aspx, which discusses various alternatives (with their pros and cons) to model a Singleton pattern.

3. **Why are you marking the instance as volatile in double checked locking example?**

   Answer:

   Let's see what C# specification tells you:

   The volatile keyword indicates that a field might be modified by multiple threads that are executing at the same time. Fields that are declared volatile are not subject to compiler optimizations that assume access by a single thread. This ensures that the most up-to-date value is present in the field at all times.

   In simple terms, the `volatile` keyword can help you to provide a serialize access mechanism. In other words, all threads will observe the changes by any other thread as per their execution order. You will also remember that the `volatile` keyword is applicable for class (or struct) fields; you cannot apply it to local variables.

4. **Why are multiple object creations a big concern?**

   Answer:

   - Object creations in the real world are treated as costly operations.
   - Sometimes you may need to implement a centralized system for easy maintenance. This also helps you to provide a global access mechanism.

5. **Why are you using the keyword "sealed"? The singleton class has a private constructor that can stop the derivation process. Is the understanding correct?**

   Answer:

   Good catch. It was not mandatory but it is always better to show your intention clearly. I have used it to guard one special case-if you are tempted to use a derived nested class as below:

```
//public sealed class Singleton
//Not using "sealed" keyword now
```

```
public class Singleton
    {
        private static readonly Singleton instance = new Singleton();
        private static int numberOfInstances = 0;
        //Private constructor is used to prevent
        //creation of instances with 'new' keyword outside this class
        //protected Singleton()
        private Singleton()
        {
            Console.WriteLine("Instantiating inside the private
            constructor.");
            numberOfInstances++;
            Console.WriteLine("Number of instances ={0}",
            numberOfInstances);
        }
        public static Singleton Instance
        {
            get
            {
                Console.WriteLine("We already have an instance
                now.Use it.");
                return instance;
            }
        }
        //The keyword "sealed" can guard this scenario.
        public class NestedDerived : Singleton { }
    }
```

Now inside `Main()` method, you can create multiple objects with statements like these:

```
Singleton.NestedDerived nestedClassObject1 =
new Singleton.NestedDerived(); //1
Singleton.NestedDerived nestedClassObject2 =
new Singleton.NestedDerived(); //2
```

So, I always prefer to use “sealed” keyword in a similar context.