

Research Project

Implementing the Singleton pattern in C#

Name: Le Nho Bach

Student ID: 103487884

Abstract:

Design Patterns are general solutions that are created for common problems in designing programs. Singleton is one of the most remarkable patterns due to its advantages and simplicity. Singleton is used to assuring that there is one and only one instance for the class and that instance provides a common access point. We can access that instance everywhere.

Introduction:

Most objects in an application are responsible for their own work and access self-contained data and references in their given scope. However, there are many objects that have additional tasks and have broader effects, such as managing limited resources or monitoring the entire state of the system. For example, on your phone there can be many apps, but there is only one operating system managing all of those apps. When we want other classes to access an instance, we need to clone and return it many times. This can make the program become slower with bad performance. Besides, it makes the design structure more sophisticated and hard to maintain. So we need a solution that allows us to create a special instance, which is unique and can be accessed everywhere in the program easily. Singleton is that solution. Singleton assures that the returned instances are all the same.

Method

So how can we ensure that there is only one instance of that class? Firstly, we need to make the constructor private, so that it cannot be created outside. Then we create another static private variable. This is the unique instance created inside the class. Finally, we provide a public static method that returns the above instance. This is the only way to access that instance and can be used everywhere to access this instance.

```
class Garden
{
    string _water;
    private static Garden GARDEN = new Garden();

    1 reference
    private Garden()
    {
        _water = "Water the plant";
    }

    2 references
    public static Garden GoToGarden()
    {
        return GARDEN;
    }

    3 references
    public void Water()
    {
        Console.WriteLine(_water);
    }

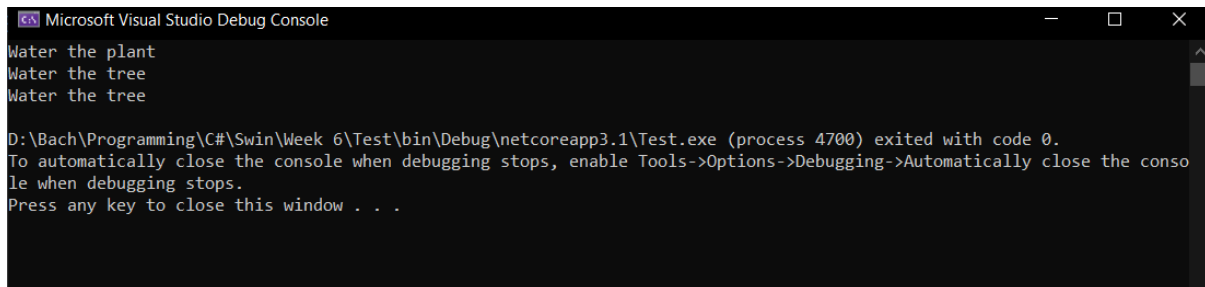
    1 reference
    public string WaterVar
    {
        get { return _water; }
        set { _water = value; }
    }
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Garden G = Garden.GoToGarden();
        G.Water();
        //We changed the WaterVar to "Water the tree"
        G.WaterVar = "Water the tree";
        G.Water();
        // We created a new Garden B, but the WaterVar is still "Water the tree"
        Garden B = Garden.GoToGarden();
        B.Water();
    }
}
```

Sample Code

Result

As we can see in the above code, the Garden instance is accessed through the method GoToGarden(). Then I changed the variable `_water` through the Property `WaterVar`. After creating a new Garden named B, the returned value when I called `Water()` is also “Water a tree” instead of “Water a plant” because both variables G and B point to the same instance inside the class Garden.



```
Microsoft Visual Studio Debug Console
Water the plant
Water the tree
Water the tree

D:\Bach\Programming\C#\Swin\Week 6\Test\bin\Debug\netcoreapp3.1\Test.exe (process 4700) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Output

There are many ways to implement Singleton:

1. Eager Initialization:

This is also the way I implemented Singleton in the previous example.

```
public class Garden
{
    private static Garden instance = new Garden();

    1 reference
    private Garden() { }

    0 references
    public static Garden getInstance()
    {
        return instance;
    }
}
```

Sample code

This is the easiest way, but it has the disadvantage that even though the instance has been initialized, it may not be used.

2. Lazy Initialization:

This way has overcome the disadvantage of Eager initialization, only when `GoToGarden()` is called will the instance be initialized. However, this way only works well in single-threaded cases, where if there are 2 threads running and calling `getInstance` at the same time, then of course we have at least 2 instances of the instance.

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

    1 reference
    private Garden() { }

    public static Garden GoToGarden()
    {
        if (instance == null)
        {
            instance = new Garden();
        }
        return instance;
    }
}
```

3. Thread-Safety:

The thread in the below code is locked on a shared object and checks to see if an instance has been created. This ensures that an instance is created by just one thread. The main issue with this is that performance decreases since a lock is required every time an instance is requested.

```
public sealed class Garden
{
    static Garden instance = null;
    static readonly object padlock = new object();
    1 reference
    Garden()
    {
    }
    0 references
    public static Garden Instance
    {
        get
        {
            lock (padlock)
            {
                if (instance == null)
                {
                    instance = new Garden();
                }
                return instance;
            }
        }
    }
    0 references
    public static string GetGardenName()
    {
        return "Thread Safe Singleton";
    }
}
```

4. Thread-Safety using Double-Check Locking:

The thread in the preceding code is locked on a shared object and uses double checking to determine whether or not an instance has been created.

```
public sealed class Garden
{
    static Garden instance = null;
    static readonly object padlock = new object();
    1 reference
    private Garden()
    {
    }
    0 references
    public static Garden Instance
    {
        get
        {
            if (instance == null)
            {
                lock (padlock)
                {
                    if (instance == null)
                    {
                        instance = new Garden();
                    }
                }
            }
            return instance;
        }
    }

    0 references
    public static string GetGardenName()
    {
        return "Garden Singleton";
    }
}
```

Discussion:

Advantages of using Singleton:

- The class only has the one and only instance
- We can access the instance everywhere → Easier to maintain
- Singleton only initializes when we call them for the first time
- It can still inherit from other classes
- It can be extended into a Factory pattern.

As we can see, Singleton is a perfect answer for the problem I mentioned above. Compared to static classes, we can see its merits as more clear and impactful:

- Static classes cannot be extended
- Static classes can still have many instances
- A Static Class cannot be initialized with a state (parameter)
- A Static class is loaded automatically by the common language runtime when the program or namespace containing the class is loaded

I found that Singleton is a good tool to control special instances which demand to be unique. Singleton can simplify my code and make it easier to update.

However, like everything, Singleton has its own drawbacks:

- Because Singleton introduces a global state into an application, it is more difficult to do unit testing
- Because an object must be serialized in a multi-threaded system to access the singleton, this style inhibits the possibility of parallelism within a program (by locking)
- Violation of the Single Responsibility Principle - the principle of a single duty, a pattern solves 2 problems at the same time.
- The Singleton pattern can hide bad design for instances when the components in the program know each other well.
- Creating too much dependence and hard to apply polymorphism.

Conclusion:

Because of its benefits and simplicity, Singleton is one of the most outstanding patterns. The term singleton is used to ensure that there is only one instance of a class and that instance offers a common access point. We have access to that instance from anywhere.

We also have many ways to implement Singleton, like Eager Initialization, Lazy Initialization, Thread-Safety, Thread-Safety using Double-Check Locking...

Reference:

Bloch, J & Garrod, C n.d., 23 Patterns in 80 Minutes: a Whirlwind Java-centric Tour of the Gang-of-Four Design Patterns, viewed 1 August 2022, <<https://www.cs.cmu.edu/~charlie/courses/15-214/2016-spring/slides/24%20-%20All%20the%20GoF%20Patterns.pdf>>.

C#Corner 2012, 'Singleton Design Pattern', www.c-sharpcorner.com, viewed <<https://www.c-sharpcorner.com/UploadFile/ff2f08/singleton-design-pattern/>>.

Viblo 2017, 'Học Singleton Pattern trong 5 phút.', Viblo, viewed <<https://viblo.asia/p/hoc-singleton-pattern-trong-5-phut-4P856goOKY3>>.