28. Singleton

Singleton design pattern allows you to use a single instance of a particular object throughout your application. This is achieved with the help of a private constructor and a static method that can call this constructor from the outside.

Singleton can be summarized as follows:

- A **Singleton** class that is meant to be used as a single instance that is shared throughout an application has a private constructor, so it can only be created inside of the class itself.
- The class holds a private static field containing its own data type.
- To instantiate the class from the outside, the class has a static method that returns the same data type as the class itself.
- This method will create an instance of the class via the private constructor and populate the private static field with this instance. Then this instance would be returned to the caller.
- Any subsequent calls will just return the object instance that was already created.

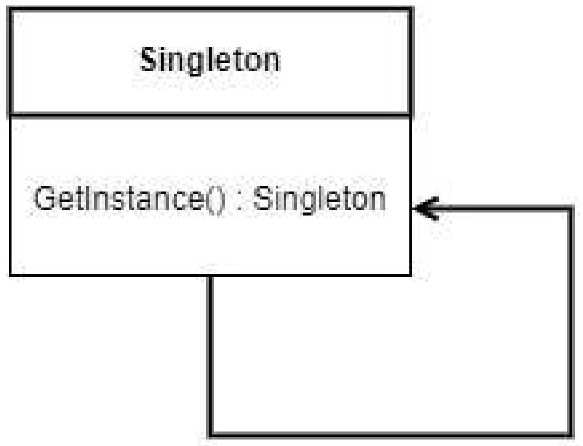


Figure 28.1 - Singleton UML diagram

We will now go through an example implementation of Singleton. The complete solution can be found via the link below:

https://github.com/fiodarsazanavets/design-patterns-in-csharp/tree/main/Creational_Patterns/Singleton

Prerequisites

In order to be able to implement the code samples below, you need the following installed on your machine:

- .NET 8 SDK (or newer)
- A suitable IDE or a code editor (Visual Studio, Visual Studio Code, JetBrains Rider)

Singleton implementation example

We will create a .NET console application and add a **Singleton** class to it. For this purpose, we will add a `SingletonObject.cs` file with the following content:

```
namespace Singleton_Demo;

internal class SingletonObject
{
    private static SingletonObject? instance;

    public static SingletonObject GetInstance()
    {
        if (instance is null)
        {
            var random = new Random();
            instance =
                new SingletonObject(random.Next());
        }

        return instance;
    }

    private SingletonObject(int data)
    {
        Data = data;
    }

    public int Data { get; private set; }
}
```

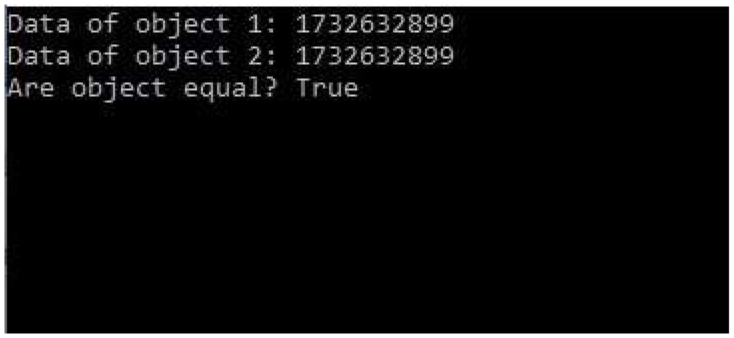
So, as you can see, we have a private constructor and a private static `instance` field with the same data type as the class itself. Then there is a static `GetInstance` method, which creates a new instance of the object on the first call and then just reuses it in every subsequent call.

Because it's a static method, it will behave exactly the same, regardless of where in the application it's called from. This means that the object instance that it will return will also be the same.

Now, we will replace the content of the Program.cs file with the following:

```
using Singleton_Demo;

var object1 = SingletonObject.GetInstance();
Console.WriteLine(
    $"Data of object 1: {object1.Data}");

var object2 = SingletonObject.GetInstance();
Console.WriteLine(
    $"Data of object 2: {object2.Data}");

Console.WriteLine(
    $"Are objects equal? {object.Equals(object1, object2)\
}");

Console.ReadKey();
```

So, essentially, we are verifying whether the same instance of an object gets returned every time we call the GetInstance method on the **Singleton** object. And, as the console output demonstrates, it does indeed return the same instance of the object:



```
Data of object 1: 1732632899
Data of object 2: 1732632899
Are object equal? True
```

Figure 28.2 - Singleton object returns the same instance of itself

And this concludes the overview of Singleton design pattern. Let's now summarize its key benefits.

Benefits of using Singleton

The key benefits of using Singleton are as follows:

- The object is only initialized once - the first time you retrieve it.
- The design pattern enforces the use of the same instance of the object throughout the application.
- The design pattern is easy to understand and set up.

However, just like any other design pattern, Singleton has its caveats. And this is what we will have a look at next.

Caveats of using Singleton

Arguably, the Singleton design pattern violates the single responsibility principle, as the **Singleton** object is responsible for both maintaining an instance of itself and performing the actual functionality the object was designed for.

Another disadvantage of using Singleton is that it may not behave as intended in a multi-threaded application. It could be that each thread would create its own instance of a **Singleton** object.

Also, the **Singleton** object isn't necessarily the easiest object type to write tests against. Especially, it will be difficult to mock, as static methods cannot be mocked by conventional means.

Finally, there is no flexibility in how the object is used. A **Singleton** object can only ever be used as **Singleton**. It cannot be used as an instance per dependency under a different context.

But all of these problems can easily be solved. The classic Singleton design pattern, as demonstrated here, has been superseded by dependency injection. Any dependency injection framework, whether it's an inbuilt ASP.NET Core system or a third-party framework, such as Autofac or Ninject, allows you to register absolutely any object as **Singleton**. Then, this object will be simply injected into the constructor of any class that depends on it.

For example, if we were to use the standard dependency injection mechanism in ASP.NET Core, all we need to do is add the following line in our `Program.cs` file before the application is built:

```
services.AddSingleton<Interface, Implementation>();
```

In this context, `Interface` represents an interface that is used as a constructor parameter. `Implementation` represents a concrete class that implements this interface that will actually be injected into the constructors. It is an example of the dependency inversion principle that was described in **chapter 5**.

This way, you will be able to easily mock your objects, so writing tests becomes easy. You no longer have to maintain private constructors and static methods, so you can maintain the single responsibility principle. Also, this allows you to have this object type as a non-singleton object in other contexts. Finally, registering a dependency as a singleton will guarantee that you will use the same instance of the object even in a multi-threaded environment.