Singletons

Object Creational Design Pattern

There can be only one...

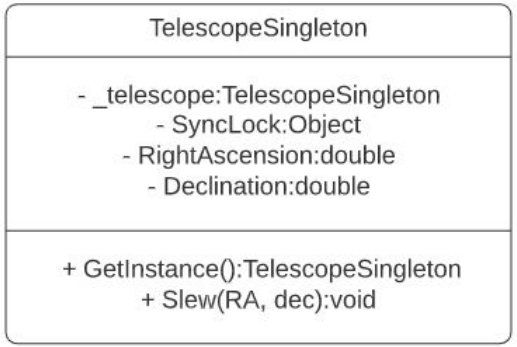
The Problem It Solves...

When we have a shared resource (like a piece of equipment or access to database) that multiple parts of our application may request access to simultaneously, this can cause issues.

For something like a logger class that's writing to a file, it doesn't make sense to instantiate a new object every time its needed. It is more efficient to create a single, shared, instance.

A singleton allows us to have an object that, during our application's lifecycle, will only ever create one instance of itself.

Implementation Details



- Static property of its own type.
- Private constructor method.
- A static "GetInstance()" method that checks if the static property is null. If it is null, it instantiates a new object (using the private constructor) and returns it. If it is not null, it returns the already instantiated object held by the static property.
- We also need to worry about different threads, as thread A could start instantiating a new singleton, and then thread B invokes GetInstance() before the static property is set. This can result in thread A and B getting different instances of the singleton.

Demo

Code Snippet

C# "lock" keyword on the SyncLock object is what allows this to be thread safe. If thread A is in that block of code, then thread B needs to wait for thread A to release the SyncLock object. SyncLock can be any object (here we are using C#'s base class, Object).

We are checking if _telescope is null twice as it's possible that another thread could instantiate the singleton while locking the SyncLock object.

```
//Singleton properties
4 references
private static TelescopeSingleton _telescope { get; set; }

private static object SyncLock = new Object();

//Telescope properties
3 references
private double RightAscension { get; set; }

3 references
private double Declination { get; set; }
```

```
//Private constructor
1 reference
private TelescopeSingleton()
{
    RightAscension = 15.0f;
    Declination = 15.0f;
}

//The publicly facing static method for getting an instance of the Singleton
2 references
public static TelescopeSingleton GetInstance()
{
    if (_telescope is null)
    {
        lock (SyncLock)
        {
            if (_telescope is null)
            {
                _telescope = new TelescopeSingleton();
            }
        }
    }
    return _telescope;
}
```

Advantages

- Managing access to a resource.
- Shared data/resources between different parts of your program.
- Potentially an improvement over a global variable as a global variable needs to be instantiated when the application starts. A singleton will only be instantiated when its GetInstance() method is called.
- Can be used like a static class, but with instance states and behavior.

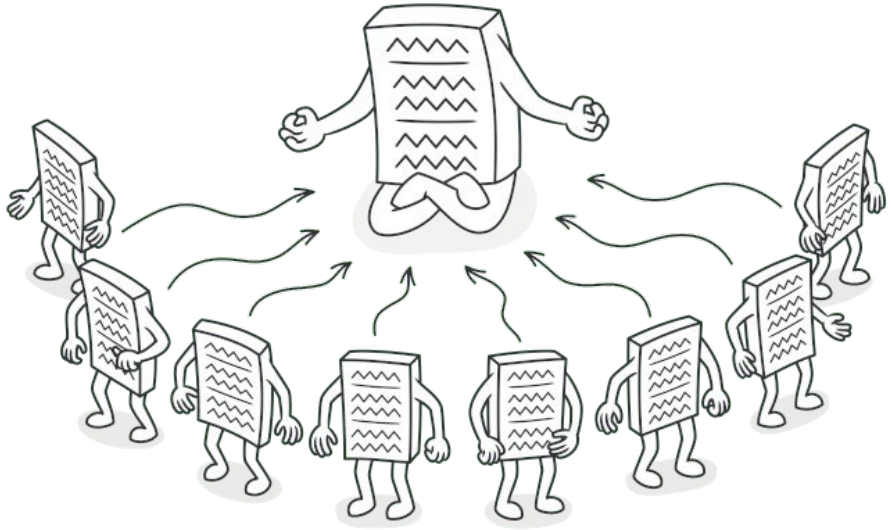
Disadvantages

- Hard to test, and almost impossible to unit test; they maintain the same state for the entire lifecycle of the application.
- Hides dependencies.
- Violates the single responsibility principle
- Tightly coupled (generally…)
- Difficult to refactor if it is later discovered you need multiple instances.

How Singletons Lost Their Groove

- Dependency injection allows us to create a single instance of an object that is shared across an entire application, but then we can still easily create other instances, if needed, or alter how we register the DI.
- Interfaces allows us to define the behavior of a class while still keeping our program loosely coupled.
- Hidden dependencies and difficult/incomplete testing makes your code more fragile.
- An arguably obsolete solution. For any problem that calls for a singleton, we can likely solve it more robustly using another design pattern.

You'll still find singletons in the wild, so you'll want to understand how they work and how to troubleshoot them.



Questions?

GitHub: https://github.com/paulmrest/SingletonDemo

Citations/Resources:

https://www.dofactory.com/net/singleton-design-pattern

https://jorudolph.wordpress.com/2009/11/22/singleton-considerations/

http://ce.sharif.edu/courses/98-99/2/ce484-1/resources/root/Design%20Patterns/Eric%20Freeman,%20Elisabeth%20Freeman,%20Kathy%20Sierra,%20Bert%20Bates-Head%20First%20Design%20Patterns%20-OReilly%20(2008).pdf

https://refactoring.guru/design-patterns/singleton