

[articles](#) [quick answers](#) [discussions](#) [features](#)

Search for articles, questions, tips

[community](#) [help](#)

Articles / Programming Languages / C#

Watch

[C#](#) [.NET](#) [singleton](#) [design-patterns](#)

Thread-Safe Singleton in C#: A Guide to Double-Checked Locking and Lazy<T> Approaches

**Antonio Ripa**

Rate me: ★★★★★ 5.00/5 (2 votes)

10 Sep 2024 CPOL 10 min read 2.9K 7 5

A practical guide to implementing the Thread-Safe Singleton pattern in C# with modern and traditional approaches, including real-world scenarios where each method excels.

This article explores the thread-safe implementation of the Singleton design pattern in C#, providing a clear comparison between the modern Lazy approach and the classic double-checked locking method. It covers the benefits and drawbacks of each, offering practical code examples and a real-world scenario where double-checked locking might still be the better choice. Whether you're implementing simple lazy initialization or need a more dynamic, flexible solution, this guide helps you choose the right approach for your application.

Introduction

Design patterns are tried-and-true solutions to common software design problems. They provide a standard way to solve recurring issues, making code more flexible, reusable, and easier to understand. This article is the first in a series exploring popular design patterns, starting with the **Singleton pattern**.

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. In this article, we'll explore how to implement a thread-safe Singleton in C#, using both the classic **double-checked locking** pattern and the more modern **Lazy<T>** approach. We'll also look at a real-world scenario where double-checked locking is a better fit.

Background

Why Use a Singleton?

The Singleton pattern guarantees that a class has exactly one instance and provides a global access point to that instance. It is often used for tasks that require a single, centralized object, like a configuration manager, a logger, or a connection pool.

Some typical use cases for the Singleton pattern include:

- **Logging:** Having a single logging object that writes to a file or console ensures consistent logging behavior.
- **Configuration Management:** A Singleton is useful for managing configuration settings that should only be initialized once during the application lifecycle.
- **Database Connections:** A Singleton can be used to manage a shared connection to a database, reducing resource overhead.

Basic Singleton Example in C#

C#



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

    // Private constructor to prevent instantiation from outside
    private Singleton() { }

    // Public static method to access the single instance
    public static Singleton Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new Singleton();
            }
            return _instance;
        }
    }

    public void DoSomething()
    {
        Console.WriteLine("Singleton instance method called!");
    }
}
```

```
}  
}
```

In this basic implementation:

- **Private Constructor:** Ensures no external classes can create new instances of **Singleton**.
- **Static Field:** Holds the single instance of the class.
- **Instance Property:** Provides global access to the instance and creates it if it doesn't already exist (lazy initialization).

How to Call the Singleton

C#

Shrink ▲ 

```
class Program  
{  
    static void Main(string[] args)  
    {  
        // Accessing the Singleton instance for the first time  
        Singleton instance1 = Singleton.Instance;  
        instance1.DoSomething();  
  
        // Accessing the Singleton instance again  
        Singleton instance2 = Singleton.Instance;  
  
        // Verifying that both references point to the same object  
        if (Object.ReferenceEquals(instance1, instance2))  
        {  
            Console.WriteLine("Both instances are the same.");  
        }  
        else  
        {  
            Console.WriteLine("Instances are different.");  
        }  
  
        // Example output:  
        // "Singleton instance method called!"  
        // "Both instances are the same."  
    }  
}
```

Here the **Singleton** class is accessed via the **Instance** property. Since the Singleton pattern ensures only one instance of the class is created, all references to the **Singleton.Instance** will point to the same object. This allows you to invoke the **DoSomething** method and ensure that the Singleton object is consistently used across your application.

- **ReferenceEquals(instance1, instance2)** checks if **instance1** and **instance2** refer to the same object in memory.
- When using the Singleton pattern, both **instance1** and **instance2** will refer to the same instance, and the program will print **"Both instances are the same."**

This demonstrates that no matter how many times the `Instance` property is called, it always returns the same object, ensuring that only one instance of the `Singleton` class exists in the application.

The Multithreading Problem

When working in multithreaded environments, ensuring that only one instance of a Singleton is created can be tricky. This is where thread-safe implementations come into play.

To illustrate the issue with multithreading in a Singleton without locks, here's an example:

Imagine you have a `StaticDataManager` class with a `GetCountries` method that fetches data from a database and populates a list of countries. Without proper synchronization, multiple threads might attempt to create the Singleton instance simultaneously, leading to unpredictable behavior or even duplicate instances.

Example Without Thread Safety

C#

Shrink ▲ 

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;

public class StaticDataManager
{
    private static ConfigurationManager _instance;
    private List<string> _countries;

    private ConfigurationManager()
    {
        _countries = new List<string>();
        LoadCountriesFromDatabase();
    }

    public static ConfigurationManager Instance
    {
        get
        {
            if (_instance == null)
            {
                _instance = new StaticDataManager();
            }
            return _instance;
        }
    }

    public List<string> GetCountries()
    {
        return _countries;
    }
}
```

```
private void LoadCountriesFromDatabase()
{
    // Here we are reading the CountryName from the database
    // and adding it to the _countries list variable _countries
}
}
```

Problem Explanation

If multiple threads call `StaticDataManager.Instance` simultaneously, there's a race condition: two or more threads may create their own instance of `StaticDataManager` before the `_instance` variable is set, causing multiple instances and redundant database calls.

This scenario can lead to:

1. **Multiple Instance Creation:** Without proper synchronization, different threads could create separate instances of `StaticDataManager`, leading to multiple database connections or unnecessary resource consumption.
2. **Redundant Database Calls:** Each thread could independently fetch the same data from the database, resulting in multiple identical queries being executed simultaneously, which increases load and latency unnecessarily.

In a real-world application, this could lead to inconsistent behavior, increased resource consumption, or incorrect data handling.

There's also another risk: what if one thread is in the process of initializing the `StaticDataManager` (e.g., it's still loading data from the database), and another thread tries to access the `GetCountries` method before the initialization is complete?

1. **Thread 1** begins the initialization process and starts loading the list of countries from the database.
2. **Thread 2** calls `GetCountries()` while **Thread 1** is still fetching data. Because **Thread 2** accesses `_countries` before it's fully populated, it could receive an incomplete list (or even an empty list if initialization hasn't added any entries yet).

In this case, the second thread might access an incomplete `_countries` list, leading to incorrect or partial data being returned. This introduces **data inconsistency**, where different parts of the application may operate on incomplete or incorrect data.

Solution

This issue is solved with proper synchronization mechanisms, such as locking, to ensure that:

- Only **one thread** can create the `StaticDataManager` instance.

- No thread can access the data (via `GetCountries`) until the initialization is fully complete.

By implementing **double-checked locking** or using `Lazy<T>`, you prevent both the race condition and the scenario where a second thread accesses the data before initialization is complete. This ensures thread-safe access to the Singleton instance and data consistency throughout the application.

The Classic Approach: Double-Checked Locking

Historically, the **double-checked locking** pattern was used to create a thread-safe Singleton in C#. The goal of this approach is to minimize the performance cost by only locking the instance creation code when necessary, i.e., during the first initialization.

Classic Singleton with Double-Checked Locking

Here's how you would implement a Singleton using double-checked locking:

C#

Shrink ▲ 

```
public class Singleton
{
    private static Singleton _instance;
    private static readonly object _lock = new object();

    private Singleton()
    {
    }

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

How It Works

- **First Null Check:** Before acquiring a lock, the code checks if the instance is already created. If it's not, the locking block is entered.

- **Locking:** If the instance hasn't been created, the code locks the critical section to ensure no other thread can create the instance simultaneously.
- **Second Null Check:** After the lock is acquired, it checks again if the instance is still `null`. This is necessary to avoid race conditions where multiple threads could pass the first check and attempt to create the instance simultaneously.

Why Use Double-Checked Locking?

Double-checked locking ensures that the Singleton instance is **lazily initialized** and **thread-safe**, with a significant focus on **performance optimization**. Here's why this method is important:

1. **Lazy Initialization:** The Singleton instance is created only when it's first needed, rather than at the start of the application, saving resources and potentially avoiding unnecessary database connections or object creation.
2. **Reduced Locking Overhead:** Without double-checked locking, the `lock` statement would be executed **every time** the `Instance` property is accessed, even after the Singleton instance is already created. This constant locking introduces unnecessary overhead, especially in high-performance or multithreaded environments where the Singleton is accessed frequently.
 - With double-checked locking, the method first checks if the instance is already created without acquiring a lock (the **first null check**). If the instance exists, it skips the lock entirely, significantly improving performance in the **common case** where the Singleton is already initialized.
3. **Thread-Safety:** During the first access when the Singleton hasn't been created, the lock ensures that only one thread initializes the instance. Once initialized, subsequent threads bypass the lock, avoiding performance penalties in future accesses.
4. **Avoiding Race Conditions:** By using two checks, you ensure that once a thread is inside the `lock` block, it rechecks the instance's state before creating it, thus preventing multiple threads from creating different instances simultaneously.

Performance Comparison:

Without double-checked locking, every access to the Singleton would require locking, which can be costly, particularly in applications with high concurrency. In contrast, with double-checked locking, once the Singleton is initialized, all future accesses avoid locking, providing **near-instantaneous** access without the overhead.

In summary, double-checked locking strikes a balance between **thread safety** and **performance** by ensuring that locking is only applied when absolutely necessary (during the first instance creation). However, in modern C# development, using `Lazy<T>` provides a simpler and more efficient way to achieve similar results.

The Modern Approach: Using `Lazy<T>`

In C# 4.0 and later, Microsoft introduced the `Lazy<T>` class, which simplifies thread-safe, lazy initialization. It is now the preferred way to implement a Singleton, as it abstracts away the complexity of locking and initialization.

Singleton Using `Lazy<T>`

C#



```
public class Singleton
{
    private static readonly Lazy<Singleton> _instance = new Lazy<Singleton>(() => new
Singleton());

    private Singleton()
    {
    }

    public static Singleton Instance => _instance.Value;
}
```

Benefits of Using `Lazy<T>`

- **Built-in Thread Safety:** `Lazy<T>` ensures that the Singleton instance is created only once, in a thread-safe manner.
- **Simplicity:** The code is significantly cleaner and easier to understand than double-checked locking.
- **Deferred Initialization:** The Singleton instance is created only when it's accessed for the first time.

Here the code for the `StaticDataManager` using the `Lazy<T>` approach

C#

Shrink ▲

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;

public class StaticDataManager
{
    // Lazy initialization of the singleton instance
    private static readonly Lazy<StaticDataManager> _instance =
        new Lazy<StaticDataManager>(() => new StaticDataManager());

    private List<string> _countries;
```



```

// Private constructor to prevent instantiation from outside
private StaticDataManager()
{
    _countries = new List<string>();
    LoadCountriesFromDatabase();
}

// Public accessor for the singleton instance
public static StaticDataManager Instance => _instance.Value;

public List<string> GetCountries()
{
    return _countries;
}

// Simulated database call to load countries
private void LoadCountriesFromDatabase()
{
    // Here we are reading the CountryName from the database
    // and adding it to the _countries list variable _countries
}
}

```

When `Lazy<T>` is Enough

For most modern applications, using `Lazy<T>` is the best way to implement a Singleton. It handles thread safety and lazy initialization out of the box, and the code is much simpler and easier to maintain.

When to Use Double-Checked Locking: A Real-World Example

Although `Lazy<T>` is ideal for most situations, there are still some cases where the **double-checked locking** approach is more appropriate. One such scenario is when you need to conditionally initialize the Singleton instance based on runtime parameters.

Scenario: Dynamic Configuration Manager

Let's consider an example where you need to load configuration settings from different sources based on the runtime environment. If the application is in development mode, the configuration is loaded from a local file; if it's in production, the configuration is loaded from a database. `Lazy<T>` doesn't allow for this kind of dynamic initialization logic, but double-checked locking does.

Here's how you could implement a dynamic configuration manager:

```
public class ConfigurationManager
{
    private static ConfigurationManager _instance;
    private static readonly object _lock = new object();

    private string _configurationSource;

    // Private constructor
    private ConfigurationManager(string configurationSource)
    {
        _configurationSource = configurationSource;
        LoadConfiguration();
    }

    private void LoadConfiguration()
    {
        if (_configurationSource == "File")
        {
            Console.WriteLine("Loading configuration from file...");
            // Load from file
        }
        else if (_configurationSource == "Database")
        {
            Console.WriteLine("Loading configuration from database...");
            // Load from database
        }
    }

    public static ConfigurationManager GetInstance(string environment)
    {
        if (_instance == null)
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    string configSource = environment == "Development" ? "File" :
"Database";
                    _instance = new ConfigurationManager(configSource);
                }
            }
        }
        return _instance;
    }
}
```

Why Use Double-Checked Locking Here?

In this case, double-checked locking gives you the flexibility to pass dynamic parameters to the Singleton instance during its initialization. This flexibility is not available with `Lazy<T>`, which assumes that the initialization logic is fixed at compile time.

Key Takeaways

- **Dynamic Initialization:** If the initialization logic depends on runtime parameters (like environment-specific configurations), double-checked locking is more flexible.
- **Control Over Initialization:** Double-checked locking allows for more complex initialization scenarios that `Lazy<T>` doesn't easily support.

Further Development

While double-checked locking and `Lazy<T>` provide reliable solutions for thread-safe Singleton initialization, further enhancements can be considered based on your application's specific needs:

1. **Dependency Injection (DI):** In modern applications, especially with frameworks like ASP.NET Core, using Dependency Injection can sometimes be a better alternative to Singletons. It ensures controlled and testable lifetime management of objects, without the global state concerns that Singletons may introduce.
2. **Eager Initialization:** In scenarios where performance is critical and the cost of instance creation is negligible, **eager initialization** can be used. This approach pre-instantiates the Singleton at application startup, avoiding any locking or lazy instantiation logic altogether. However, this is only suitable when you are certain that the Singleton will always be used.
3. **Asynchronous Lazy Initialization:** For cases where your Singleton's initialization involves async operations (such as I/O-bound tasks like database connections), you can use `Lazy<T>` combined with `async`. This allows for non-blocking, thread-safe Singleton creation in asynchronous scenarios.
4. **Custom Thread-Safe Caches:** If your Singleton manages a large amount of data, you could incorporate custom caching mechanisms (e.g., using `MemoryCache` or a thread-safe dictionary) to improve access speed and avoid unnecessary database calls.
5. **Versioned Singleton:** For advanced scenarios, you might implement a versioned Singleton where different configurations or instances are needed for various parts of the application. This can be useful in systems where a Singleton represents stateful objects, such as different database connections or configurations for different tenants.

By considering these techniques, you can adapt the Singleton pattern to better suit specific performance, concurrency, or scalability needs in your projects.

Conclusion

While `Lazy<T>` is the go-to approach for thread-safe, lazy Singleton initialization in modern C# development, there are still scenarios where double-checked locking is preferable. If your Singleton

initialization requires dynamic parameters or complex decision-making based on runtime conditions, double-checked locking provides the necessary flexibility.

In most cases, however, the simplicity and elegance of `Lazy<T>` make it the preferred choice. As always, the right solution depends on the specific needs of your application.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOl\)](#)

Written By

Antonio Ripa

Architect

 Switzerland

I really like coding and continually strive to improve my knowledge of design patterns and emerging architectures. My journey began 41 years ago, when at the age of 13, I bought my first Sinclair ZX Spectrum and started teaching myself to code. Today, I am a Senior Solution Architect with expertise in Agile methodologies, specializing in the design and development of Enterprise Applications across both back-end and front-end. I have extensive experience working in the Fintech and Medtech domains.

Over the years, I have developed a keen interest in Enterprise Design Patterns, Domain Driven Design, Test Driven Design, and Scrum Methodology. I'm always on the lookout for new patterns that can enhance software development practices. I am also passionate about AI, leveraging machine learning and artificial intelligence technologies to optimize solutions and automate processes within enterprise systems. My AI experience ranges from integrating intelligent algorithms into applications to enhancing user experiences with predictive models.

I believe knowledge should be shared, so I actively contribute to the coding community through platforms like Code Project, sharing insights that may benefit other software engineers and developers in this incredible industry. I don't claim to have all the answers, but the practices I've adopted have played a significant role in my own career. If they resonate with you, feel free to try them. And if you have any comments—positive or constructive—I'd love to hear from you.



Watch

This is a Collaborative Group

10 members
















Apply to join this group

Comments and Discussions

Add a Comment or Question  Email Alerts 

Spacing **Relaxed**  Layout **Normal**  Per page **25**  Update

First Prev Next

 Do NOT use DCL 	 wkempf	10-Sep-24 21:53
 Re: Do NOT use DCL 	 Antonio Ripa	11-Sep-24 2:05
 Re: Do NOT use DCL 	 wkempf	11-Sep-24 2:28
 Re: Do NOT use DCL 	 Antonio Ripa	11-Sep-24 3:57
 Re: Do NOT use DCL 	 wkempf	10hrs 13mins ago

Last Visit: 10-Sep-24 16:27 Last Update: 12-Sep-24 8:53 Refresh **1**

 General  News  Suggestion  Question  Bug  Answer  Joke  Praise  Rant 

Admin

Use Ctrl+Left/Right to switch messages, Ctrl+Up/Down to switch threads, Ctrl+Shift+Left/Right to switch pages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Layout: [fixed](#) | [fluid](#)
Posted 10 Sep 2024

Article Copyright 2024 by Antonio Ripa
Everything else Copyright ©
[CodeProject](#), 1999-2024
Web04 2.8:2024-07-22:1