

C# Singleton Design Pattern - Complete Deep Dive

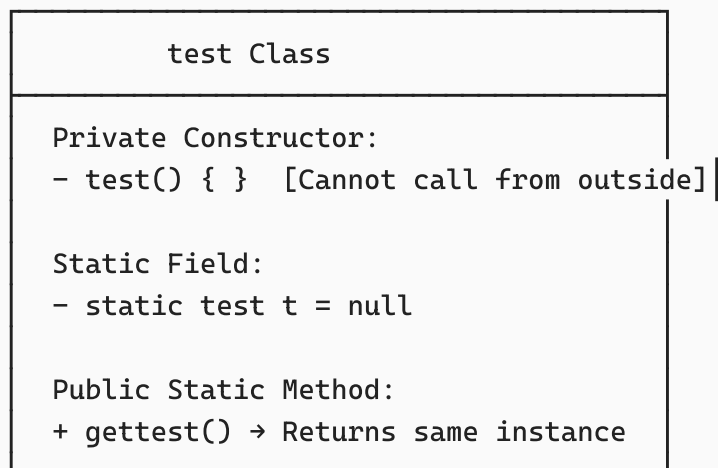
1. What is the Singleton Pattern?

The **Singleton Pattern** is a creational design pattern that ensures a class has **only one instance** throughout the application lifetime and provides a **global point of access** to that instance.

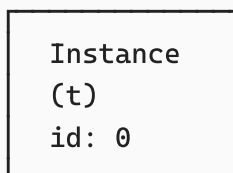
Core Concepts:

- **One Instance Only:** No matter how many times you request an object, you always get the same instance
- **Global Access:** Accessible from anywhere in the application
- **Lazy Initialization:** Instance created only when first needed (optional)
- **Private Constructor:** Prevents external instantiation

Visual Representation:



↓



↑

All references point
to THIS single object

2. Analyzing the Code

Example from code:

```
class test
{
    public int id { get; set; }

    test(int id=0) // Private constructor (no access modifier)
    {
        this.id = id;
    }

    static test t; // Static field to hold the single instance

    public static test gettest() // Factory method
    {
        if (t == null)
        {
            t = new test();
            return t;
        }
        else
            return t;
    }
}
```

Breaking Down Each Component:

1. Private Constructor

```
test(int id=0)
{
    this.id = id;
}
```

Characteristics:

- No access modifier means **private by default** for constructors
- Prevents external code from creating instances
- Only accessible from within the class itself

What it prevents:

```
// ❌ This won't compile
test t = new test(); // Error: 'test.test(int)' is inaccessible due to its
protection level
```

2. Static Instance Field

```
static test t;
```

Characteristics:

- **Static** - belongs to the class, not to any instance
- Initialized to `null` by default
- Holds the **single shared instance**
- Lives for the entire application lifetime
- Accessible from the static method

3. Factory Method (gettest)

```
public static test gettest()
{
    if (t == null)
    {
        t = new test();
        return t;
    }
    else
        return t;
}
```

Logic flow:

1. Check if instance exists (`t == null`)
2. If **null** (first call): Create new instance, store it, return it
3. If **not null** (subsequent calls): Return existing instance
4. Result: Always returns the same instance

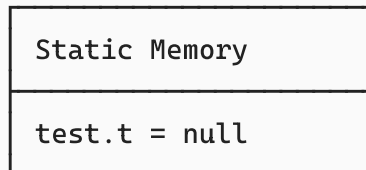
3. Execution Flow Analysis

Example from code:

```
test t1 = test.gettest();
test t2 = test.gettest();
Console.WriteLine(t1.GetHashCode());
Console.WriteLine(t2.GetHashCode());
```

Step-by-Step Execution:

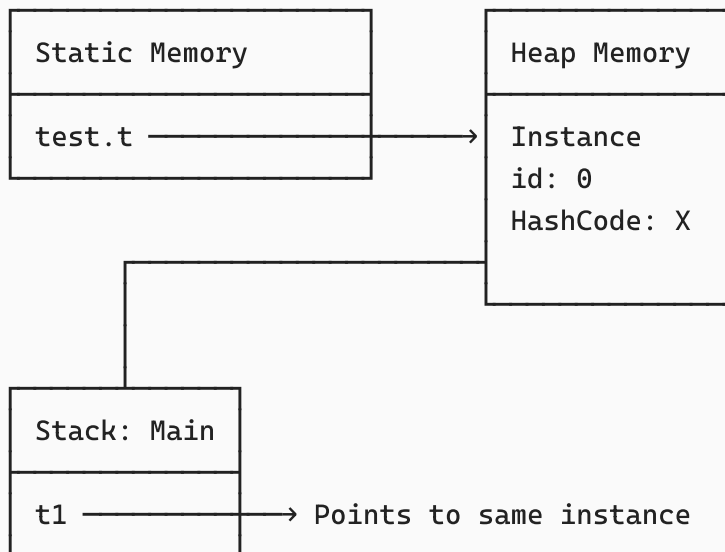
Initial State:



Step 1: test t1 = test.gettest();

1. Call gettest()
2. Check: t == null? → YES
3. Create: t = new test()
4. Return t

After Step 1:

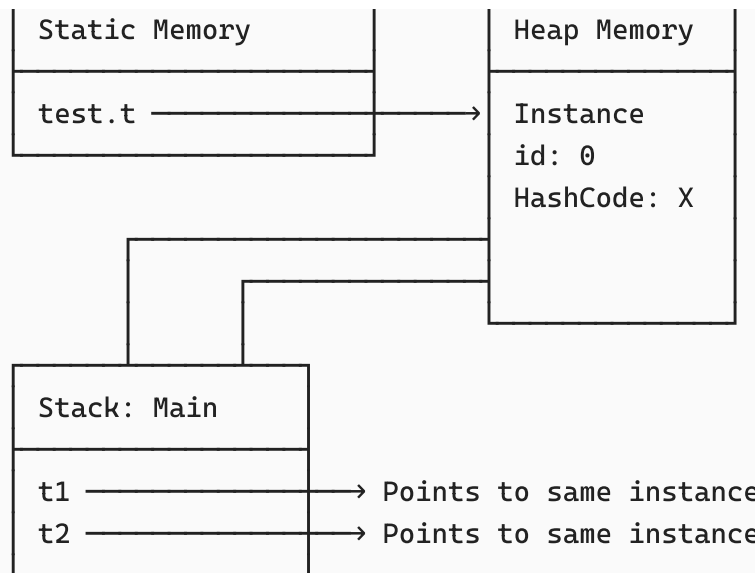


Step 2: test t2 = test.gettest();

1. Call gettest()
2. Check: t == null? → NO (already exists)
3. Return existing t

After Step 2:





Step 3: GetHashCode()

```
Console.WriteLine(t1.GetHashCode()); // Output: e.g., 58225482
Console.WriteLine(t2.GetHashCode()); // Output: e.g., 58225482 (SAME!)
```

Why HashCodes Are Identical:

GetHashCode() returns a hash code based on the **object's memory address** (by default). Since **t1** and **t2** reference the **same object** in memory, they have the **same hash code**.

```
Console.WriteLine(t1.GetHashCode()); // 58225482
Console.WriteLine(t2.GetHashCode()); // 58225482 (identical)

// Proof they're the same object:
Console.WriteLine(ReferenceEquals(t1, t2)); // True
Console.WriteLine(t1 == t2);                // True (same reference)

// Modifying through one reference affects the other:
t1.id = 10;
Console.WriteLine(t2.id); // 10 (same object!)
```

4. Why Use the Singleton Pattern?

Use Cases:

1. Database Connection Manager

Only one connection pool should exist:

```

class DatabaseManager
{
    private static DatabaseManager instance;
    private SqlConnection connection;

    private DatabaseManager()
    {
        connection = new SqlConnection(connectionString);
    }

    public static DatabaseManager GetInstance()
    {
        if (instance == null)
        {
            instance = new DatabaseManager();
        }
        return instance;
    }
}

```

2. Configuration Settings

Single source of configuration:

```

class AppSettings
{
    private static AppSettings instance;
    public string Theme { get; set; }
    public string Language { get; set; }

    private AppSettings()
    {
        // Load settings from file
    }

    public static AppSettings GetInstance()
    {
        if (instance == null)
        {
            instance = new AppSettings();
        }
        return instance;
    }
}

```

```
// Usage everywhere:  
var settings = AppSettings.GetInstance();  
string theme = settings.Theme;
```

3. Logger

Centralized logging:

```
class Logger  
{  
    private static Logger instance;  
    private StreamWriter logFile;  
  
    private Logger()  
    {  
        logFile = new StreamWriter("app.log", append: true);  
    }  
  
    public static Logger GetInstance()  
    {  
        if (instance == null)  
        {  
            instance = new Logger();  
        }  
        return instance;  
    }  
  
    public void Log(string message)  
    {  
        logFile.WriteLine($"{DateTime.Now}: {message}");  
        logFile.Flush();  
    }  
}  
  
// Usage:  
Logger.GetInstance().Log("Application started");
```

4. Cache Manager

Single cache instance:

```
class CacheManager  
{  
    private static CacheManager instance;  
    private Dictionary<string, object> cache;
```

```
private CacheManager()
{
    cache = new Dictionary<string, object>();
}

public static CacheManager GetInstance()
{
    if (instance == null)
    {
        instance = new CacheManager();
    }
    return instance;
}

public void Set(string key, object value)
{
    cache[key] = value;
}

public object Get(string key)
{
    return cache.ContainsKey(key) ? cache[key] : null;
}
}
```

5. Singleton Implementation Variations

5.1 Lazy Initialization (From Code)

Current implementation:




```
class test
{
    static test t;

    public static test gettest()
    {
        if (t == null)
        {
            t = new test();
        }
        return t;
    }
}
```



```
}  
}
```

Characteristics:

-  Instance created only when first needed
-  Saves memory if never used
-  **Not thread-safe** (problem in multi-threaded apps)

5.2 Thread-Safe Singleton (Lock)

```
class test  
{  
    private static test t;  
    private static readonly object lockObject = new object();  
  
    private test(int id = 0)  
    {  
        this.id = id;  
    }  
  
    public static test gettest()  
    {  
        if (t == null)  
        {  
            lock (lockObject)  
            {  
                if (t == null) // Double-check locking  
                {  
                    t = new test();  
                }  
            }  
        }  
        return t;  
    }  
}
```

Why double-check?

Thread 1: Checks `t == null` → true → enters lock
Thread 2: Checks `t == null` → true → waits for lock
Thread 1: Inside lock, creates instance, exits
Thread 2: Gets lock, checks AGAIN (`t` now `!= null`), doesn't create
Result: Only one instance created





5.3 Eager Initialization

```
class test
{
    // Created immediately when class is loaded
    private static readonly test t = new test();

    private test(int id = 0)
    {
        this.id = id;
    }

    public static test gettest()
    {
        return t; // Just return pre-created instance
    }
}
```

Characteristics:

-  Thread-safe by default (CLR guarantees)
-  Simple implementation
-  Instance created even if never used
-  Can't handle initialization exceptions well

5.4 Lazy Implementation (Modern .NET)

```
class test
{
    // Lazy<T> handles thread-safety automatically
    private static readonly Lazy<test> lazy =
        new Lazy<test>(() => new test());





    private test(int id = 0)
    {
        this.id = id;
    }

    public static test Instance => lazy.Value;
}

// Usage:
```

```
test t1 = test.Instance;  
test t2 = test.Instance;
```

Characteristics:

-  Thread-safe
-  Lazy initialization
-  Clean, modern syntax
-  Best of both worlds

5.5 Property-Based Singleton

```
class test  
{  
    private static test t;  
    private static readonly object lockObject = new object();  
  
    private test(int id = 0)  
    {  
        this.id = id;  
    }  
  
    public static test Instance  
    {  
        get  
        {  
            if (t == null)  
            {  
                lock (lockObject)  
                {  
                    if (t == null)  
                    {  
                        t = new test();  
                    }  
                }  
            }  
            return t;  
        }  
    }  
}  
  
// Usage (more elegant):  
test t1 = test.Instance;  
test t2 = test.Instance;
```

6. Thread Safety Issues

The Problem with Lazy Singleton (From Code):

```
// ❌ NOT THREAD-SAFE
public static test gettest()
{
    if (t == null)           // Thread 1 checks
    {                         // Thread 2 checks at same time
        t = new test();      // Both create instances!
        return t;
    }
    return t;
}
```

Race condition scenario:

Time	Thread 1	Thread 2	t value
1	if (t == null) → true		null
2		if (t == null) → true	null
3	t = new test()		Instance A
4		t = new test()	Instance B ❌

Result: Two instances created! Singleton broken!

Solution: Thread-Safe Implementation

```
// ✅ THREAD-SAFE
private static readonly object lockObject = new object();


public static test gettest()
{
    if (t == null)
    {
        lock (lockObject) // Only one thread at a time
        {
            if (t == null) // Double-check
            {
                t = new test();
            }
        }
    }
}
```

```

    }
    return t;
}













```

With lock:

Time	Thread 1	Thread 2	t value
1	if (t == null) → true		null
2	lock acquired		null
3	if (t == null) → true		null
4	t = new test()		Instance A
5	lock released		Instance A
6		if (t == null) → false	Instance A
7		return t	Instance A 

7. Comparison with Other Patterns

Singleton vs Static Class:

Feature	Singleton	Static Class
Instantiation	One instance	No instances
Inheritance	 Can inherit/implement interfaces	 Cannot inherit
Lazy initialization	 Possible	 Members initialized on first access
Polymorphism	 Can use as interface type	 No polymorphism
State	 Can have instance state	 Only static state
Testing	 Harder to mock	 Very hard to mock
Dependency Injection	 Can be injected	 Cannot inject

Example:

```

// Singleton can implement interfaces
interface ILogger
{
    void Log(string message);
}

```

```

}

class Logger : ILogger
{
    private static Logger instance;
    private Logger() { }

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

    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}

// Can be used polymorphically
ILogger logger = Logger.Instance;

// Static class cannot implement interfaces
static class StaticLogger
{
    public static void Log(string message)
    {
        Console.WriteLine(message);
    }
}

```

8. Advantages and Disadvantages

Advantages:

1. **Controlled Access:** Single point of control over the instance
2. **Memory Efficiency:** Only one instance in memory
3. **Global State:** Accessible from anywhere
4. **Lazy Initialization:** Resource created only when needed

5. **Namespace Pollution Prevention:** Better than global variables

Disadvantages:

1. **Testing Difficulties:** Hard to mock for unit tests
 2. **Hidden Dependencies:** Classes depend on singleton implicitly
 3. **Global State Issues:** Can lead to tight coupling
 4. **Thread Safety Complexity:** Requires careful implementation
 5. **Violates Single Responsibility:** Controls instance creation + business logic
 6. **Difficult to Subclass:** Private constructor prevents inheritance
-

9. Modern Alternatives

Dependency Injection (Preferred in Modern .NET):

Instead of:

```
// ❌ Old way - tight coupling
class OrderService
{
    public void ProcessOrder()
    {
        var logger = Logger.GetInstance();
        logger.Log("Processing order");
    }
}
```

Use DI:

```
// ✅ Modern way - loose coupling
class OrderService
{
    private readonly ILogger _logger;

    public OrderService(ILogger logger) // Injected
    {
        _logger = logger;
    }

    public void ProcessOrder()
    {

```

```
        _logger.Log("Processing order");
    }
}

// In Startup.cs or Program.cs
services.AddSingleton<ILogger, Logger>(); // Framework manages singleton
```

Benefits:

- Testable (can inject mock)
 - Loosely coupled
 - SOLID principles
 - Framework handles thread safety
-

10. Best Practices

✓ DO:

1. **Make constructor private**
2. **Use thread-safe implementation** for multi-threaded apps
3. **Consider Lazy** for modern C#
4. **Document why singleton is necessary**
5. **Implement IDisposable** if managing resources

✗ DON'T:

1. **Overuse singletons** (anti-pattern if misused)
2. **Store mutable state** without thread safety
3. **Use for everything** (violates dependency injection principles)
4. **Ignore testing difficulties**

Modern Recommendation:

```
// Best practice: Use Lazy<T>
class DatabaseManager
{
    private static readonly Lazy<DatabaseManager> lazy =
        new Lazy<DatabaseManager>(() => new DatabaseManager());

    public static DatabaseManager Instance => lazy.Value;
```



```
private DatabaseManager()
{
    // Initialize
}

// Or even better: Use DI container
}
```

Key Takeaways Summary

1. **Singleton Pattern:** Ensures only one instance of a class exists
2. **Private Constructor:** Prevents external instantiation
3. **Static Instance:** Holds the single shared instance
4. **Factory Method:** Provides global access point (e.g., `GetInstance()`)
5. **Lazy Initialization:** Instance created on first access
6. **Thread Safety:** Critical in multi-threaded applications
7. **GetHashCode():** Same for all references (proves same object)
8. **Use Cases:** Database connections, loggers, configurations, caches
9. **Lazy:** Modern, thread-safe implementation
10. **DI Alternative:** Preferred in modern .NET applications
11. **Testing:** Singletons are hard to test; prefer dependency injection
12. **Global State:** Convenient but can lead to tight coupling

The Singleton pattern is **powerful but should be used sparingly**. In modern C# development, **dependency injection** with singleton lifetime is often a better choice!

By Abdullah Ali

Contact : +201012613453