

Online Training

- C#.NET Online Training Program
- ASP.NET Core Training
- Microservices Online Training using .NET Core
- Microsoft Azure Training

Introduction & Environment Setup

- How Computer Works
- Introduction to Programming Languages
- How Computer Programs Works
- Different Types of Applications
- Programming Methodologies
- Algorithm, Pseudocode, Programs, and Flowcharts
- Introduction to .NET Framework
- .NET Framework Architecture and Components
- Introduction to C#
- Programming Language
- How to Download and Install Visual Studio on Windows
- Creating First Console Application using Visual Studio
- .NET Developer Roadmap for 2025
- Coding Standard Best Practices

C#/.NET Basics

- Basic Structure of C# Program
- Methods and Properties of Console Class in C#
- Data Types in C#
- Literals in C#
- Type Casting in C#
- Variables in C#
- Operators in C#
- Control Flow Statements in C#
- If-Else Statements in C#
- Switch Statements in C#
- Loops in C#
- While Loop in C#
- Do While Loop in C#
- For Loop in C#
- Break Statement in C#
- Continue Statement in C#
- Goto Statement in C#
- Functions in C#
- User-Defined Functions in C#
- Call By Value and Call By Reference in C#
- Recursion in C#
- User Input and Output in C#
- Command Line Arguments in C#
- String in C#
- Static Keyword in C#
- Static vs Non-Static Members in C#
- Const and Read-Only in C#
- Properties in C#
- Why we Should Override ToString Method in C#
- Override Equals Method in C#
- Difference Between Convert.ToString and ToString Method in C#
- Checked and Unchecked Key words in C#
- Stack and Heap Memory in .NET
- Boxing and Unboxing in C#

OOPs in C#

- Object Oriented Programming (OOPs) in C#
- Class and Objects in C#
- Constructors in C#
- Types of Constructors in C#
- Why We Need Constructors in C#
- Static vs Non-Static Constructors in C#
- Private Constructors in C#
- Destructors in C#
- Garbage Collection in .NET Framework
- Differences Between Finalize and Dispose in C#
- Access Specifiers in C#
- Encapsulation in C#
- Abstraction in C#
- Inheritance in C#
- Types of Inheritance in C#
- How to use Inheritance in Application Development
- IsA and HasA Relationship in C#
- Generalization and Specialization in C#
- Abstract Class and Abstract Methods in C#
- Abstract Class and Abstract Methods Interview Questions in C#
- How to Use Abstract Classes and Methods in C#
- Application
- Interface in C#
- Interface Interview Questions and Answers in C#
- Interface Realtime Examples in C#
- Multiple Inheritance in C#
- Multiple Inheritance Realtime Example in C#
- Polymorphism in C#
- Method Overloading in C#
- Operator Overloading in C#
- Method Overriding in C#
- Method Hiding in C#
- Partial Class and Partial Methods in C#
- Sealed Class and Sealed Methods in C#
- Extension Methods in C#
- Static Class in C#
- Variable Reference and Instance of a Class in C#

OOPs Real-Time Examples

- Real-Time Examples of Encapsulation Principle in C#
- Real-Time Examples of Abstraction Principle in C#
- Real-Time Examples of Inheritance Principle in C#
- Real-Time Examples of Polymorphism Principle in C#
- Real-Time Examples of Interface in C#
- Real-Time Examples of Abstract Class in C#

Exception Handling

- Exception Handling in C#
- Multiple Catch Blocks in C#
- Finally Block in C#
- How to Create Custom Exceptions in C#
- Inner Exception in C#
- Exception Handling Abuse in C#

Events, Delegates and Lambda Expression in C#

- Course Structure of Events, Delegates and Lambda Expression
- Roles of Events, Delegates and Event Handler in C#
- Delegates in C#
- Multicast Delegates in C#
- Delegates Real-Time Example in C#
- Generic Delegates in C#
- Anonymous Method in C#
- Lambda Expressions in C#
- Events in C# with Examples

Multi-Threading

- Multithreading in C#
- Thread class in C#
- How to Pass Data to Thread Function in Type Safe Manner in C#
- How to Retrieve Data from a Thread Function in C#
- Join Method and Joinive Property of Thread Class in C#
- Thread Synchronization in C#
- Lock in C#
- Monitor Class in C#
- Mutex Class in C#
- Semaphore Class in C#
- SemaphoreSlim Class in C#
- Performance Testing of a Multithreaded Application
- Thread Pool in C#
- Foreground and Background Threads in C#
- AutoResetEvent and ManualResetEvent in C#
- Thread Life Cycle in C#
- Threads Priorities in C#
- How to Terminate a Thread in C#
- Inter Thread Communication in C#
- How to Debug a Multi-threaded Application in C#

Collections in C#

- Arrays in C#
- 2D Arrays in C#
- Advantages and Disadvantages of Arrays in C#
- Collections in C#
- ArrayList in C#
- Hashtable in C#
- Non-Generic Stack in C#
- Non-Generic Queue in C#
- Non-Generic SortedList in C#
- Advantages and Disadvantages of Non-Generic Collection in C#
- Generic Collections in C#
- Generics in C#
- Generic Constraints in C#
- Generic List Collection in C#
- How to Sort a List of Complex Type in C#
- Comparison Delegate in C#
- Dictionary Collection Class in C#
- Conversion Between Array List and Dictionary in C#
- List vs Dictionary in C#
- Generic Stack Collection Class in C#
- Generic Queue Collection Class in C#
- Foreach Loop in C#
- Generic HashSet Collection Class in C#
- Generic SortedSet Collection Class in C#
- Generic SortedSet Collection Class in C#
- Generic SortedDictionary Collection Class in C#
- Generic LinkedList Collection Class in C#
- Concurrent Collection in C#
- ConcurrentDictionary Collection Class in C#
- ConcurrentQueue Collection Class in C#
- ConcurrentStack Collection Class in C#
- ConcurrentBag Collection Class in C#
- BlockingCollection in C#

File Handling

- File Handling in C#
- FileStream Class in C#
- StreamReader and StreamWriter in C#
- File Class in C#
- TextWriter and TextReader in C#
- BinaryWriter and BinaryReader in C#
- StringReader and StringWriter in C#
- FileInfo Class in C#
- DirectoryInfo Class in C#
- Export and Import Excel Data in C#

Asynchronous Programming

- Introduction to Concurrency
- Async and Await in C#
- Task in C#
- How to Return a Value from Task in C#
- How to Execute Multiple Tasks in C#
- How to Limit Number of Concurrent Tasks in C#
- How to Cancel a Task in C#
- How to Create Synchronous Method using Task in C#
- Only Pattern in C#
- Retry One Pattern in C#
- How to Control the Result of a Task in C#
- Task-Based Asynchronous Programming in C#
- Continuation Tasks
- How to Attached Child Tasks to a Parent Task in C#
- ValueTask in C#
- How to Cancel a Non-Cancellable Task in C#
- Asynchronous Streams in C#
- How to Cancel Asynchronous Stream in C#

Parallel Programming

- Task Parallel Library in C#
- Parallel For in C#
- Parallel Foreach Loop in C#
- Parallel Invoke in C#
- Maximum Degree of Parallelism in C#
- How to Cancel Parallel Operations in C#
- Atomic Methods Thread Safety and Race Conditions in C#
- Interlocked vs Lock in C#
- Parallel LINQ in C#
- Multithreading vs Asynchronous Programming vs Parallel Programming in C#

AutoMapper

- AutoMapper in C#
- AutoMapper Complex Mapping in C#
- How to Map Complex Type to Primitive Type using AutoMapper in C#
- AutoMapper Reverse Mapping in C#
- AutoMapper Conditional Mapping in C#
- AutoMapper Ignore Method in C#
- Fixed and Dynamic Values in Destination Property in AutoMapper

Optional Parameter, Indexers and Enums

- How to make Optional Parameters in C#
- Indexers Real-Time Example in C#
- Enums in C#

.NET Framework Architecture

- DOT NET Framework
- Common Language Runtime in .NET Framework
- .NET Program Execution Process
- Intermediate Language (ILDASM & ILASM) Code in C#
- Common Type System in .NET Framework
- Common Language Specification in .NET Framework
- Managed and Unmanaged Code in .NET Framework
- Assembly DLL EXE in .NET Framework
- App Domain in .NET Framework
- Strong and Weak Assemblies in .NET Framework
- How to Install an Assembly into GAC in .NET Framework
- DLL Hell Problem and Solution in .NET Framework

Var, Dynamic and Reflection

- Reflection in C#
- Dynamic Type in C#
- Var Keyword in C#
- Var vs Dynamic in C#
- Dynamic vs Reflection in C#
- Volatile Keyword in C#
- Ref vs Out in C#
- Named Parameters in C#

C# 7.X New Features

- C# 7 New Features
- Enhancement in Out Variables in C# 7
- Pattern Matching in C#
- Digit Separators in C# 7
- Tuples in C# 7
- Splitting Tuples in C# 7
- Local Functions in C# 7
- Ref Returns and Ref Locals in C# 7
- Generalized Async Return Types in C# 7
- Expression Bodied Members in C#
- Thrown Expression in C#
- Async Main in C#

C# 8 New Features

- C# 8 New Features
- ReadOnly Structs in C#
- Default Interface Methods in C#
- Pattern Matching in C#
- Using Declarations in C#
- Static Local Functions in C#
- Disposable Ref Structs in C#
- Nullable Reference Types in C# 8
- Asynchronous Streams in C#
- Asynchronous Disposable in C#
- Indices and Assignment Operator in C#
- Null-Coalescing Assignments in C#
- Unmanaged Constructed Types in C#
- Stackalloc in C#

Most Popular C# Books

- Most Recommended C# Books
- Most Recommended Data Structure and Algorithms Books using C#

Lock in C#

Back to: [C#.NET Tutorials For Beginners and Professionals](#)

Thread Synchronization Using Lock in C#

In this article, I am going to discuss **Thread Synchronization using Lock in C#** with Examples. Please read our previous article where we discussed **Thread Synchronization in C#** with Examples.

1. **Why do we need to Protect the Shared Resources in Multithreading in C#?**
2. **Accessing a Shared Resource in a Single-Threaded Environment in C#**
3. **Accessing a Shared Resource in a Multithreaded Environment in C#**
4. **How to Protect a Shared Resource in a Multithread Environment from Concurrent Access in C#?**
5. **What is the Lock Statement in C#?**
6. **How is the Lock Statement internally work in C#?**
7. **Protecting Shared Variable using Lock Statement in C# with Example**

Why do we need to Protect the Shared Resources in Multithreading in C#?

In a multithreading application, it is very important for us to handle multiple threads for executing critical section code or you can say shared resources. For example, if we have a shared resource, and multiple threads want to access the shared resource, then we need to protect the shared resource from concurrent access otherwise we will get some inconsistent behavior or output.

In C#, we can use Lock and Monitor to provide thread safety in a multithreaded application. Both lock and monitor provide a mechanism that ensures that only one thread is executing the critical section code at any given point in time to avoid any functional breaking of code or to avoid inconsistent behavior or output.

In this article, I am going to discuss how to protect the shared resource in a multithread environment using the lock and in the next article, I am going to discuss how to do the same thing using the [Monitor in C#](#).

Accessing a Shared Resource in a Single-Threaded Environment in C#:

Before understanding how to use the lock to protect the shared resource in a multithread environment in C#, let us first understand the problem if we are not protecting the shared resource in a multithread environment. In the following example, we have a shared resource i.e. DisplayMessage() method and we call that method three times from the Main method as shown below.

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            DisplayMessage();
            DisplayMessage();
            DisplayMessage();

            Console.Read();
        }

        static void DisplayMessage()
        {
            Console.WriteLine("Welcome to the ");
            Thread.Sleep(1000);
            Console.WriteLine("world of dotnet!");
        }
    }
}
```

```
[Welcome to the [Welcome to the [Welcome to the world of dotnet!]
world of dotnet!]
world of dotnet!]
world of dotnet!]
```

As the above program is a single-threaded program, so we got the output as expected. Here, the shared resource is the DisplayMessage method which is going to be executed three times sequentially, and hence we got the correct output. Now, let us proceed and see what happens if we access the shared resources in a multithreaded environment.

Accessing a Shared Resource in a Multithreaded Environment in C#:

In the following example, we have created three different threads and then invoke the same DisplayMessage() method using all these three different threads. Here, DisplayMessage() method is the shared resource and this shared resource is simultaneously or concurrently invoked by three different threads. Here, we are not protecting the shared resource, and all three threads accessing the shared resource which results in inconsistent output.

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t1 = new Thread(DisplayMessage);
            Thread t2 = new Thread(DisplayMessage);
            Thread t3 = new Thread(DisplayMessage);

            t1.Start();
            t2.Start();
            t3.Start();

            Console.Read();
        }

        static void DisplayMessage()
        {
            Console.WriteLine("Welcome to the ");
            Thread.Sleep(1000);
            Console.WriteLine("world of dotnet!");
        }
    }
}
```

```
[Welcome to the [Welcome to the [Welcome to the world of dotnet!]
world of dotnet!]
world of dotnet!]
world of dotnet!]
```

As you can see, here we are not getting the output as expected. So, the point that you need to keep in mind is that if the shared resource is not protected in a multithreaded environment from concurrent access, then the output or the behavior of the application becomes inconsistent.

How to Protect a Shared Resource in a Multithread Environment from Concurrent Access in C#?

We can protect the shared resources in a multithread environment from concurrent access by using the concept Monitor and Locking in C#. Let us see how to protect the shared resource using the lock statement in C# and see the output. In the following example, we will discuss the **Monitor Class Enter and Exit Methods** in detail, for now, to understand, what we can say is, it acquires an exclusive lock within the try block by invoking the Monitor class Enter method and it releases the exclusive lock within the finally block by calling the Monitor class Exit method.

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static void Main(string[] args)
        {
            Thread t1 = new Thread(DisplayMessage);
            Thread t2 = new Thread(DisplayMessage);
            Thread t3 = new Thread(DisplayMessage);

            t1.Start();
            t2.Start();
            t3.Start();

            Console.Read();
        }

        private static readonly object LockDisplayMethod = new object();
        static void DisplayMessage()
        {
            lock(LockDisplayMethod)
            {
                Console.WriteLine("Welcome to the ");
                Thread.Sleep(1000);
                Console.WriteLine("world of dotnet!");
            }
        }
    }
}
```

Now run the application and see the output as expected as shown below. As you can see in the below output, we are getting the result as expected and this is because we are now protecting the shared resource using the lock statement which will ensure that only one thread will be able to access the critical section at any given point in time.

```
[Welcome to the world of dotnet!]
[Welcome to the world of dotnet!]
[Welcome to the world of dotnet!]
```

What is the lock statement in C#?

According to [Microsoft](#), the lock statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released.

Note: When you want to synchronize thread access to a shared resource, you should lock the shared resource on a dedicated object instance (for example, **private readonly object _lockObject = new object();**) or **private static readonly object _lockObject = new object();**. Avoid using the same lock object instance for different shared resources, as it might result in a deadlock.

How is the lock Statement internally work in C#?

The lock statement in C# was internally converted to a try-finally block when we compiled the code. The compiled code of the lock statement will look like the below. You can see, it is internally using the Monitor class Enter and Exit method. In our next article, we will discuss the **Monitor Class Enter and Exit Methods** in detail, for now, to understand, what we can say is, it acquires an exclusive lock within the try block by invoking the Monitor class Enter method and it releases the exclusive lock within the finally block by calling the Monitor class Exit method.

```
bool lockWasTaken = false;
var temp = new object();
try
{
    Monitor.Enter(temp, ref lockWasTaken);
    // body
}
finally
{
    if (lockWasTaken)
    {
        Monitor.Exit(temp);
    }
}
```

You can also verify the IL Code by using the ILDASM tool. Now, if you open the application using the ILDASM tool and if you verify the DisplayMessage method IL Code, then you can see the try and finally blocks along with the Monitor class Enter and Exit methods as shown in the below image.

```
.method private hidebysig static void DisplayMessage() cil managed
{
    // Code size 67 (0x43)
    .maxstack 2
    .locals init ([8] object 0_0, [1] bool 0_1)
    IL_0000: nop
    IL_0001: ldstr object ThreadingDemo.Program::LockDisplayMethod
    IL_0006: stloc.0
    IL_0007: ldc.i4.0
    IL_0008: stloc.1
    try
    {
        IL_0009: ldloc.0
        IL_000a: ldloc.5
        IL_000c: call void [mscorlib]System.Threading.Monitor::Enter(object, bool)
        IL_0011: nop
        IL_0012: nop
        IL_0013: ldstr ""[Welcome to the "
        IL_0018: call void [mscorlib]System.Console::Write(string)
        IL_001d: nop
        IL_001e: ldc.i4 0x3e8
        IL_0023: call void [mscorlib]System.Threading.Thread::Sleep(int32)
        IL_0028: nop
        IL_0029: ldstr "world of dotnet!]"
        IL_002e: call void [mscorlib]System.Console::WriteLine(string)
        IL_0033: nop
        IL_0034: nop
        IL_0035: leave.s IL_0042
    } // end try
    finally
    {
        IL_0037: ldloc.1
        IL_0038: brfalse.s IL_0041
        IL_003a: ldloc.0
        IL_003b: call void [mscorlib]System.Threading.Monitor::Exit(object)
        IL_0040: nop
        IL_0041: endfinally
    } // end of method Program::DisplayMessage
}
```

Protecting Shared Variable using Lock Statement in C# with Examples:

The section or block or particular resource that you want to protect should be placed inside the lock block. Let us understand this with an example. In the below example, we are only protecting the shared Count variable from concurrent access.

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static int Count = 0;

        static void Main(string[] args)
        {
            Thread t1 = new Thread(IncrementCount);
            Thread t2 = new Thread(IncrementCount);
            Thread t3 = new Thread(IncrementCount);

            t1.Start();
            t2.Start();
            t3.Start();

            //Wait for all three threads to complete their execution
            t1.Join();
            t2.Join();
            t3.Join();

            Console.WriteLine(Count);
            Console.Read();
        }

        static void IncrementCount()
        {
            for (int i = 1; i <= 1000000; i++)
            {
                //Only protecting the shared Count variable
                lock (LockCount)
                {
                    Count++;
                }
            }
        }
    }
}
```

When you run the above program, it will give you the output as expected as 3000000. Now, let us see what happens if we are not protecting our shared variable Count. In the below example, we are not protecting the Count shared variable and hence all three threads simultaneously access the variable and try to increment the value, and hence we will get some unexpected output.

```
using System.Threading;
using System;
namespace ThreadingDemo
{
    class Program
    {
        static int Count = 0;

        static void Main(string[] args)
        {
            Thread t1 = new Thread(IncrementCount);
            Thread t2 = new Thread(IncrementCount);
            Thread t3 = new Thread(IncrementCount);

            t1.Start();
            t2.Start();
            t3.Start();

            //Wait for all three threads to complete their execution
            t1.Join();
            t2.Join();
            t3.Join();

            Console.WriteLine(Count);
            Console.Read();
        }

        static void IncrementCount()
        {
            for (int i = 1; i <= 1000000; i++)
            {
                Count++;
            }
        }
    }
}
```

Every time, you run the application, you will get a different output. So, it is important for us to protect our shared resources in a multithreaded environment, or else we will not get the expected output.


Here, in this article, I try to explain **Thread Synchronization using Lock in C#** with Examples. In the next article, I am going to discuss **How to Protect Shared Resources in a Multithread Environment using Monitor Class in C#** with Examples. I hope you enjoy this Thread Synchronization using Lock in C# with Examples article.



Dot Net Tutorials

About the Author: Pranaya Rout

Pranaya Rout has published more than 3,000 articles in his 11-year career. Pranaya Rout has very good experience with Microsoft Technologies, including C#, VB, ASP.NET MVC, ASP.NET Web API, EF, EF Core, ADO.NET, LINQ, SQL Server, MySQL, Oracle, ASP.NET Core, Cloud Computing, Microservices, Design Patterns and still learning new technologies.



Previous Lesson

Thread Synchronization in C#

Next Lesson

Monitor Class in C#

3 thoughts on "Lock in C#"



NISHANT

JULY 17, 2021 AT 6:06 PM

Good, Keep it up.

Reply



G

NOVEMBER 18, 2022 AT 5:22 PM

Great article. Actually usefull examples

Reply



LEOPOLDO

APRIL 22, 2023 AT 4:33 PM

Congrats! I like it!

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name*

Email*

Website

Post Comment

About Us

Privacy Policy

Contact

ADO.NET Tutorial

Angular Tutorials

ASP.NET Core Blazor Tutorials

ASP.NET Core Tutorials

ASP.NET MVC Tutorials

ASP.NET Web API Tutorials

C Tutorials

C#.NET Programs Tutorials

C#.NET Tutorials

Cloud Computing Tutorials

Data Structures and Algorithms Tutorials

Design Patterns Tutorials

DotNet Interview Questions and Answers

Core Java Tutorials

Entity Framework Tutorials

JavaScript Tutorials

LINQ Tutorials

Python Tutorials

SOLID Principles Tutorials

SQL Server Tutorials

Traveling Tutorials

JBDC Tutorials

Java Servlets Tutorials

Java Struts Tutorials

C++ Tutorials

JSP Tutorials

MySQL Tutorials

Oracle Tutorials

ASP.NET Core Web API Tutorials

HTML Tutorials

© Dot Net Tutorials | Website Design by Sunrise Pixel

