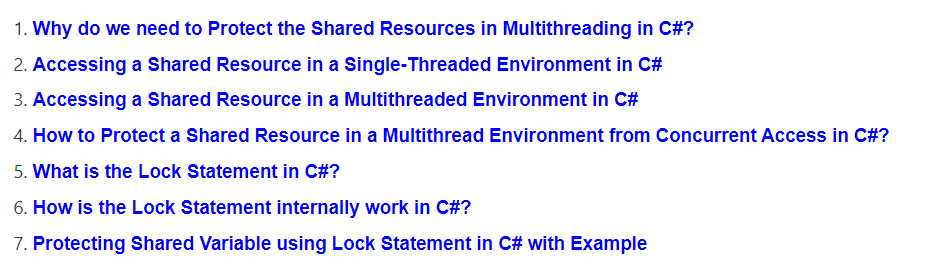
**Lock in C#**



**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.
* So , In C#, we can use Lock and Monitor to provide *thread safety in a multithreaded* *application*.
* Because 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.

##### ****Problem For Accessing a Shared Resource in a Single-Threaded Environment****

1. let us 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.



1. 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.
2. 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 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.

##### Moreover here, we are not protecting the shared resource, so all three threads accessing the shared resource will results in inconsistent output



1. As we can see, that here we are not getting the output as expected.
2. 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.*

##### ****Protecting Shared Resource in a Multithread Environment from Concurrent Access****

##### 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 have created one readonly object i.e LockDisplayMethod, and then we created a block using the lock keyword.

##### To the lock keyword, we pass the LockDisplayMethod object, and *the section or block or particular resource that we want to protect should be placed inside the lock block* (which is shown in the below example.)

##### 

##### Now run the application then we will see the output as expected i.e. 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.*

##### Output

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

##### ****What is the lock statement****

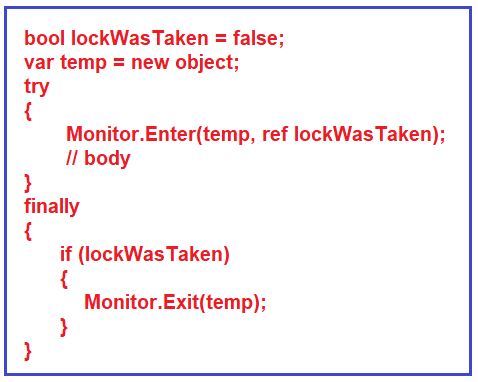
In simpler terms, the **lock** statement in C# helps manage access to shared resources among multiple threads. Here's how it works:

1. **Acquiring the Lock**: When a thread wants to access a shared resource, it first tries to acquire the lock associated with that resource. If the lock is available (i.e., no other thread currently holds it), the thread acquires the lock and proceeds to execute the code inside the **lock** block.
2. **Executing the Code**: Once the lock is acquired, the thread can safely execute the code inside the **lock** block. This ensures that no other threads can access the shared resource simultaneously, preventing conflicts and ensuring data integrity.
3. **Releasing the Lock**: After the thread has finished executing the code inside the **lock** block, it releases the lock. This allows other threads to acquire the lock and access the shared resource if needed.
4. **Blocking Other Threads**: While a thread holds the lock, any other threads that try to acquire the same lock are blocked. They wait until the lock is released before attempting to acquire it again.
5. **Avoiding Deadlocks**: To prevent deadlocks, where two or more threads are blocked indefinitely, it's important to use a dedicated lock object for each shared resource. This ensures that threads don't accidentally block each other when trying to access different resources.

Overall, the **lock** statement help us in managing access to shared resources in a way that ensures *thread safety and prevents conflicts between multiple threads*.

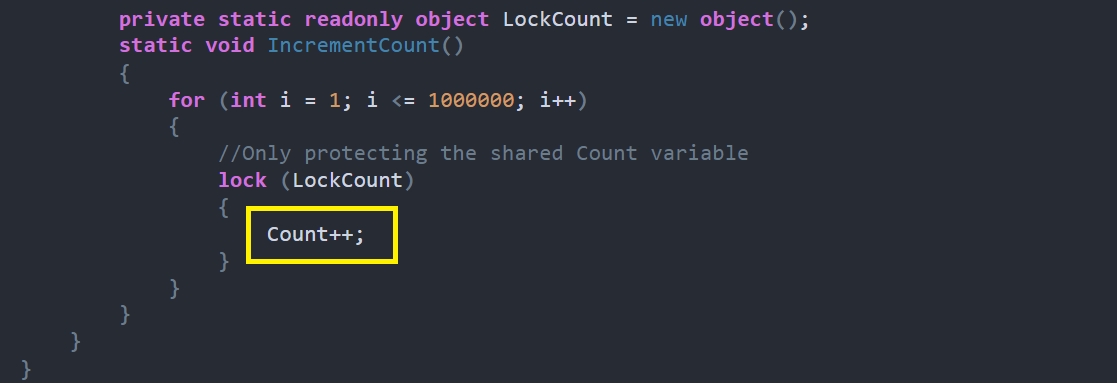
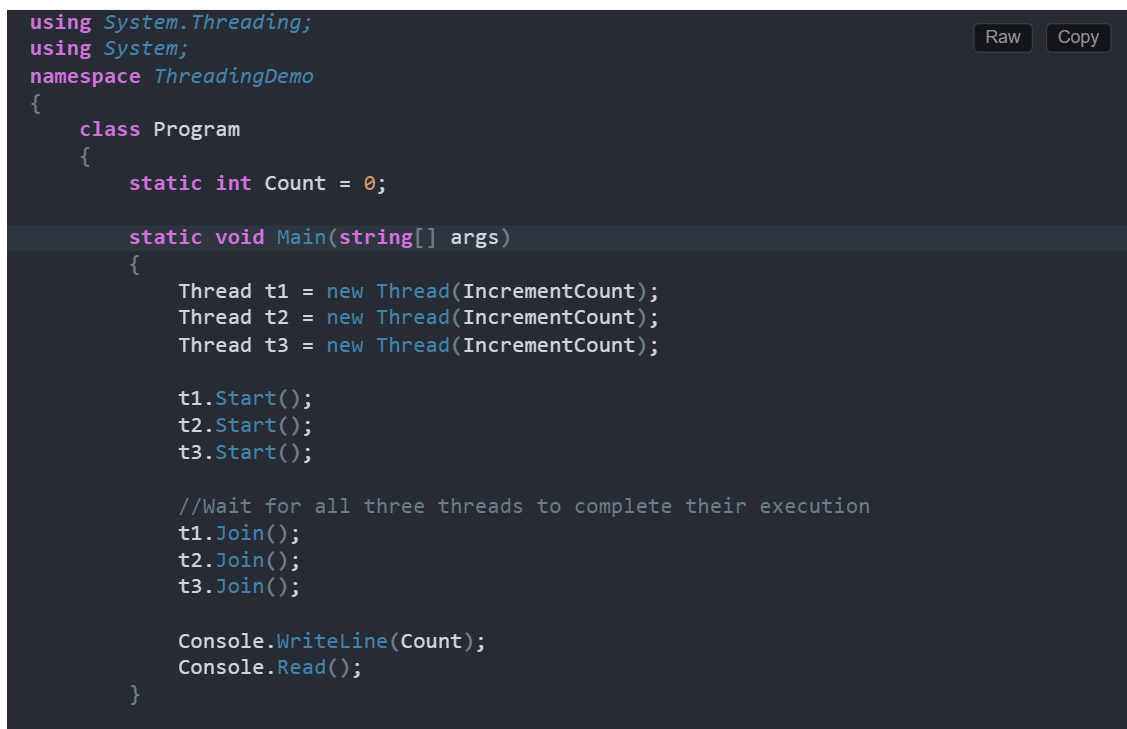
##### ****How is the lock Statement internally work ?****

1. The lock statement in C# was internally converted to a try-finally block when we compiled the code.
2. The compiled code of the lock statement will look like the below.
3. You can see, it is internally **using the Monitor class Enter and Exit method**.,
4. so it acquires an exclusive lock within the try block by invoking the Monitor class Enter method and then,
5. it releases the exclusive lock within the finally block by calling the Monitor class Exit method.



##### ****Protecting Shared Variable using Lock Statement****

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



1. When you run the above program, it will give you the output as **expected as 3000000.**
2. 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 .
3. So, all three threads simultaneously access the variable and try to increment the value, and hence we will get some *unexpected output.*



1. 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 application, or else we will not get the expected output.

Lock(this) v/s PVRO

Using **lock(this)** and **private static readonly object LockDisplayMethod = new object();** achieve similar results in terms of providing *a synchronization mechanism to prevent multiple threads from executing the critical section of code simultaneously*. However, there are some key differences and considerations:

1. **Scope**:
   * **lock(this)** locks on the current instance of the class. This means that if you have multiple instances of the class, each instance will have its own lock, and threads accessing different instances will not be blocked by each other.
   * **private static readonly object LockDisplayMethod = new object();** uses a static object as the lock. This means that all instances of the class share the same lock, and threads accessing different instances will be blocked by each other.
2. **Visibility**:
   * **lock(this)** locks are public, meaning that any code that has access to the instance can potentially lock on it. This can lead to unintended locking behavior and possible deadlocks if other code also locks on the same instance.
   * **private static readonly object LockDisplayMethod = new object();** uses a private static object for locking, which limits the visibility of the lock to just the class itself. This reduces the risk of unintended locking behavior by other parts of the code.
3. **Performance**:
   * **lock(this)** has a slight performance overhead compared to using a dedicated private static object for locking because locking on **this** involves locking on the instance itself, which is a more heavyweight operation.
   * **private static readonly object LockDisplayMethod = new object();** has less overhead because it locks on a static object, which is typically more efficient than locking on an instance.

Usage : When To Use Which

1. **Use lock(this) when**:
   * You are sure that only one instance of the class will ever exist.
   * You need per-instance locking behavior, where each instance of the class has its own lock.
   * You are aware of the potential risks and have carefully reviewed the code to ensure that no other code locks on the same instance, avoiding deadlocks.
2. **Use a dedicated private static object for locking (private static readonly object LockObject = new object();) when**:
   * You want to synchronize access to a shared resource across all instances of the class.
   * You want to ensure that only one thread can access the critical section of code across all instances of the class at a time.
   * You want to reduce the risk of unintended locking behavior by limiting the visibility of the lock to just the class itself.

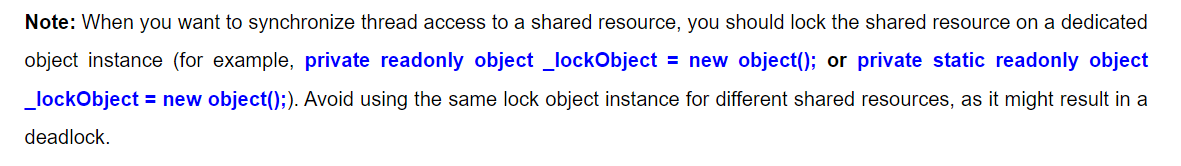
In general, it's a good practice to use a dedicated private static object for locking to avoid potential issues with visibility and unintended locking. However, if you need per-instance locking behavior and are aware of the risks, you can use **lock(this)** in a single-instance scenario.

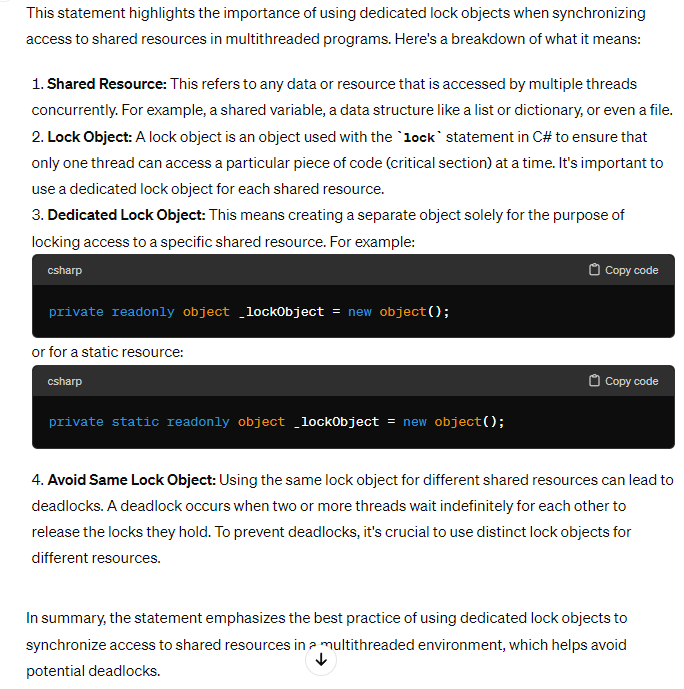
*Note:*

Using **lock(this)** can lead to unexpected behavior and is generally discouraged. The **lock** statement in C# is used to ensure that only one thread can enter a critical section of code at a time. When you use **lock(this)**, you're locking on the current instance of the class, which means that if another part of your code also locks on the same instance, you can get into a deadlock situation where both threads are waiting for each other to release the lock.

It's considered a better practice to use a dedicated object for locking, like in your example where **LockDisplayMethod** is used. This way, you avoid potential issues with locking on **this** and make your code more robust.

*Remember*





**Meaning Of this PSRO**

By using **private static readonly object \_lockObject = new object();**, we're declaring a private static readonly object named **\_lockObject** and initializing it with a new instance of an object. This object will be used as a **lock** for the **lock** statement.

Break down the keywords used in the **private static readonly object \_lockObject = new object();** statement:

1. **private**: This keyword is an access modifier that specifies that the **\_lockObject** field can only be accessed within the containing class and not from outside the class.
2. **static**: This keyword indicates that **\_lockObject** is a static member of the class, meaning there will be only one instance of **\_lockObject** shared among all instances of the class.
3. **readonly**: This keyword indicates that the value of **\_lockObject** cannot be changed once it is assigned a value. In this case, **\_lockObject** is initialized with a new object and will always refer to that object.
4. **object**: This is the type of the **\_lockObject** field. It specifies that **\_lockObject** is a reference to an instance of the **object** class.
5. **\_lockObject**: This is the name of the field. It is a convention in C# to prefix private fields with an underscore (**\_**).
6. **new object()**: This part of the statement initializes **\_lockObject** with a new instance of the **object** class. This object will be used as a lock for the **lock** statement to control access to the protected block of code.