Synchronizing data for multithreading

When multiple threads can make calls to the properties and methods of a single object, it is critical that those calls be synchronized. Otherwise one thread might interrupt what another thread is doing, and the object could be left in an invalid state. A class whose members are protected from such interruptions is called **thread-safe**. .NET provides several strategies to synchronize access to instance and static members: Synchronized code regions, Manual synchronization, Synchronized contexts and Collection classes in the System.Collections.Concurrent namespace.

Synchronized code regions

You can use the **Monitor** class or a compiler keyword to synchronize blocks of code, instance methods, and static methods. There is no support for synchronized static fields.

Both Visual Basic and C# support the marking of blocks of code with a particular language keyword, the **`lock`** statement in C# or the SyncLock statement in Visual Basic. When the code is executed by a thread, an attempt is made to acquire the lock. If the lock has already been acquired by another thread, the thread blocks until the lock becomes available. When the thread exits the synchronized block of code, the lock is released, no matter how the thread exits the block.

Protecting shared resources in multithreaded environment using `lock` statement

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. The `lock` statement is of the form

```
lock (x)
{
    // Your code...
}
```

where `x` is an expression of a reference type.

When you synchronize thread access to a shared resource, lock on a dedicated object instance (for example`, private readonly object balanceLock = new object();`) or another instance that is unlikely to be used as a lock object by unrelated parts of the code. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using the following as lock objects:

- `this`, as it might be used by the callers as a lock.
- Type instances, as those might be obtained by the typeof operator or reflection.
- `string` instances, including string literals, as those might be interned.

Hold a lock for as short time as possible to reduce lock contention.

Example:

```
using System;
using System.Threading;
public class Account
{
    private readonly object balanceLock = new object();
    private decimal balance;
    public Account(decimal initialBalance) => balance = initialBalance;
    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit
amount cannot be negative.");
        }
        decimal appliedAmount = 0;
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }
    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The credit
amount cannot be negative.");
        }
        lock (balanceLock)
        {
            balance += amount;
        }
    }
    public decimal GetBalance()
    {
        lock (balanceLock)
        {
            return balance;
        }
    }
}
class AccountTest
{
    static void Main()
    {
        var account = new Account(1000);
        Thread t1 = new Thread(Update);
        Thread t2 = new Thread(Update);
        Thread t3 = new Thread(Update);
        t1.Start(account);
        t2.Start(account);
        t3.Start(account);
        //Wait for all three threads to complete their execution
        t1.Join();
        t2.Join();
        t3.Join();
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 1030
    }
    static void Update(object obj)
    {
        var account = (Account)obj;
        decimal[] amounts = { 0, 2, -3, 6, -2, -1, 8, -5, 11, -6 };
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

Protecting shared resources in multithreaded environment using `Monitor`

The Monitor class in C# provides a mechanism that synchronizes access to objects. Let us simplify the above definition. In simple words, we can say that, like the lock, we can also use this class to protect shared resources in a multi-threaded environment. This can be done by acquiring an exclusive lock on the object so that only one thread can enter into the critical section at any given point of time.

The Monitor is a static class and belongs to the System.Threading namespace. As a static class, it provides a collection of static methods as shown below. Using these static methods you can provide access to the monitor associated with a particular object.

1. `public static void Enter (object obj);`
   Acquires an exclusive lock on the specified object.
2. `public static void Enter (object obj, ref bool lockTaken);`
   Acquires an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
3. `public static void Exit (object obj);`
   Releases an exclusive lock on the specified object.

4. `public static void Pulse (object obj);`
   Notifies a thread in the waiting queue of a change in the locked object's state.
5. `public static void PulseAll (object obj);`
   Notifies all waiting threads of a change in the object's state.
6. `public static bool TryEnter (object obj);`
   Attempts to acquire an exclusive lock on the specified object.
7. `public static void TryEnter (object obj, ref bool lockTaken);`
   Attempts to acquire an exclusive lock on the specified object, and atomically sets a value that indicates whether the lock was taken.
8. `public static bool TryEnter (object obj, int millisecondsTimeout);`
   Attempts, for the specified number of milliseconds, to acquire an exclusive lock on the specified object.
9. `public static bool Wait (object obj);`
   Releases the lock on an object and blocks the current thread until it reacquires the lock.
10. `public static bool Wait (object obj, int millisecondsTimeout);`
    Releases the lock on an object and blocks the current thread until it reacquires the lock. If the specified time-out interval elapses, the thread enters the ready queue.

The critical section

Use the Enter and Exit methods to mark the beginning and end of a critical section. If the critical section is a set of contiguous instructions, then the lock acquired by the Enter method guarantees that only a single thread can execute the enclosed code with the locked object. In this case, It's recommended that you place that code in a try block and place the call to the Exit method in a finally block. This ensures that the lock is released even if an exception occurs. The following code fragment illustrates this pattern.

```
// Define the lock object.
var obj = new Object();
// Define the critical section.
Monitor.Enter(obj);
try {
   // Code to execute one thread at a time.
}
// catch blocks go here.
finally {
   Monitor.Exit(obj);
}
```

Example:

```
using System;
using System.Threading;
namespace MonitorDemo
{
    class Program
    {
        static readonly object lockObject = new object();
        public static void PrintNumbers()
        {
            Console.WriteLine(Thread.CurrentThread.Name + " Trying to enter into
the critical section");
            Boolean IsLockTaken = false;
            Monitor.Enter(lockObject, ref IsLockTaken);
            try
            {
                Console.WriteLine(Thread.CurrentThread.Name + " Entered into the
critical section");
                for (int i = 0; i < 5; i++)
                {
                    Thread.Sleep(100);
                    Console.Write(i + ",");
                }
                Console.WriteLine();
            }
            finally
            {
                Monitor.Exit(lockObject);
                Console.WriteLine(Thread.CurrentThread.Name + " Exit from
critical section");
            }
        }
        static void Main(string[] args)
        {
            Thread[] Threads = new Thread[3];
            for (int i = 0; i < 3; i++)
            {
                Threads[i] = new Thread(PrintNumbers);
                Threads[i].Name = "Child Thread " + i;
            }
            foreach (Thread t in Threads)
            {
                t.Start();
            }
            Console.ReadLine();
        }
    }
}
// Output:
//    Child Thread 0 Trying to enter into the critical section
//    Child Thread 0 Entered into the critical section
//    Child Thread 1 Trying to enter into the critical section
//    Child Thread 2 Trying to enter into the critical section
//    0,1,2,3,4,
//    Child Thread 0 Exit from critical section
//    Child Thread 1 Entered into the critical section
//    0,1,2,3,4,
//    Child Thread 1 Exit from critical section
//    Child Thread 2 Entered into the critical section
//    0,1,2,3,4,
//    Child Thread 2 Exit from critical section
```

Difference between **`Monitor`** and **`lock`**

The functionality provided by the Enter and Exit methods of Monitor class is identical to that provided by the `lock` statement in C# except that the language construct wrap the `Monitor.Enter(Object, Boolean)` method overload and the `Monitor.Exit` method in a `try`…`finally` block to ensure that the monitor is released.

So, the `lock` provides the basic functionality to acquire an exclusive lock on a synchronized object. But, If you want more control to implement advanced multithreading solutions using `TryEnter() Wait(), Pulse(),` and `PulseAll()` methods, then the `Monitor` class is your option.